Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories

Guantao Liu, Tim Schmidt and Rainer Dömer
University of California, Irvine

Ajit Dingankar and Desmond Kirkpatrick
Intel Corporation

Abstract— With the increasing demand for parallel computing power, manycore platforms are attracting more and more attention due to their potential to improve performance and scalability of parallel applications. However, as the core count increases, core-to-core communication becomes expensive. For manycore architectures using directory-based cache coherence protocols, the core-to-core communication latency depends not only on the physical placement on the chip, but also on the location of the distributed cache tag directory. In this paper, we first define the concept of *core distance* for multicore and manycore architectures. Using a ping-pong spin-lock benchmark, we quantify the core distance on a ring-network platform and propose an approach to optimize thread-to-core mapping in order to minimize on-chip communication overhead. In our experiments, our approach speeds up communication-intensive benchmarks by more than 25% on average over the Linux default mapping strategy.

I. Introduction

Manycore processors have become popular in recent years to provide capable platforms for those highly parallel applications which have extraordinary scaling and vector capabilities that cannot be satisfied by conventional multicore processors. Generally speaking, manycore platforms refer to processors with dozens to hundreds and soon thousands of cores on chip. Thus, in order to fully utilize manycore platforms, an application must scale well past hundreds of threads and distribute equal workloads among those parallel threads.

One recent example of a manycore processor is the Xeon Phi™ coprocessor, a readily available implementation of the Intel® Many Integrated Core (MIC) architecture. Fig. 1 depicts the conceptual structure of the Intel Xeon Phi architecture [1]. On the single die of the coprocessor, up to 61 x86-based cores are integrated. These parallel processing cores communicate via a high performance bidirectional ring interconnect. Each core is fully functional and fetches and decodes instructions in-order from four hardware thread contexts. Thus, in total, there are 240 logical cores available on the Intel Xeon Phi 5110P coprocessor. For the on-chip cache hierarchy, each core includes 32 KB L1 instruction and data cache, as well as a private 512 KB L2 cache. In order to keep the L2 cache data globally consistent and reduce hot-spot contention for data references, a distributed Tag Directory (TD) is coresident with each core to cross-snoop L2 caches in all cores. Every physical memory address is uniquely mapped to one of 64 distributed tag directories on the ring network via a reversible hash function. Using the distributed TD infrastructure, the caches are kept consistent without software intervention. In general, the Xeon Phi coprocessor can be viewed as a symmetric multiprocessor (SMP) with shared Uniform Memory Access (UMA) [2][3].

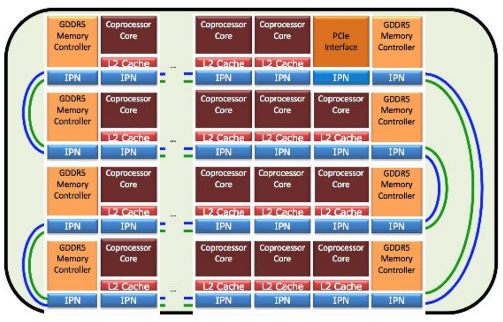


Fig. 1. Intel® Xeon Phi™ coprocessor architecture [1].

A. Motivation

Increasing the number of cores provides potential to improve performance and scalability for highly parallel programs, but also brings downsides. As the chip size is enlarged to accomodate more processing units, core-to-core transfers are not always significantly better than main memory latency and optimization becomes crucial. For the Xeon Phi coprocessor specifically, because of the opaque hashing method and the resulting ”random” distribution of addresses around the ring, no previous software optimization has been found to improve core-to-core transfers significantly [3]. In the remainder of this paper, we propose our software strategy to optimize thread-to-core mapping on manycore platforms with distributed tag directories, and show that it minimizes core-to-core communication latency.

II. Core Distance

In this section, we define core distance and provide a memory ping-pong benchmark to quantify it. We then show measured core distances for a multicore and manycore processor to show the architectural differences between these platforms.

A. Definition of Core Distance

Each core usually has its own local cache to utilize program locality and speed up memory access. In order to keep the data globally coherent among all caches, the sharing of a data block is broadcast on the interconnecting medium (in snooping protocols) or passed to the directory that tracks the state of the cache (in directory-based protocols). By maintaining coherent caches, data can be easily transferred via shared variables between cores. However the core-to-core transfer latency varies a lot between cores on the same processor and different processors. To quantify the core-to-core tranfer latency, we can define *core distance* as the duration it takes to move a data word back and forth between two cores (one round trip). A larger core distance then means that it is more expensive to share data between these two cores.

Algorithm 1 Ping-Pong Communication Function

```
1:  cycles_t pingpong ($varptr$, $val$)
2:  { $T_1$ := CurrentCycles()
3:    for $iter = 1$ to ITERATION do
4:      while Load($varptr$) $= val$ do
5:        Processor Pause
6:      end while
7:      Store($varptr$, $val$)
8:    end for
9:    $T_2$ := CurrentCycles()
10:   return $(T_2 - T_1)$/ITERATION
11: }
```

In order to measure core distances on multicore and manycore processors, we propose a memory ping-pong communication benchmark. Fig. 2 shows the communication between two cores and Algorithm 1 lists the **pingpong** function executing in the threads, which are bound to the measured cores. Each thread starts a timer in local variable T_1. Then, it compares the shared memory address *varptr* against its local value *val*. If they match, the program goes into the while loop, pauses the core for a few clock cycles and then checks the sharing address again. Otherwise, the thread stores its *val* to the sharing address which is then communicated through the cache hierarchy to the other core. Each thread tries to update the shared variable in turn and runs this procedure for multiple iterations. At the end, the program stops the timer, and returns the average round-trip communication latency.

B. Core Distances on Hierarchical Multicore Platforms

As expected, on multicore systems using snooping cache coherency, core distance is highly correlative to the core placement on the platform. For example, Fig. 3 illustrates the architecture of a Intel® Xeon dual-CPU system. Fig. 4 shows the corresponding measured core distances from Core 0 to other cores. As hyperthreading is enabled, there are 32 logical cores in total on this platform. In Fig. 4, the core distance between the two logical threads on the same physical core (Core 0 and 16) is minimum as the two hyperthreads can communicate via the local L1 or L2 cache. Since Core 0 to 7 are on the same processor and Core 8 to 15 are on the other one, interprocessor communication is more expensive, costing 1200 vs. 300 cycles for a round trip.

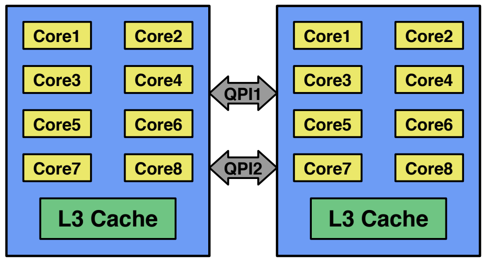


Fig. 3. Intel® Xeon E5-2680 dual-CPU architecture.

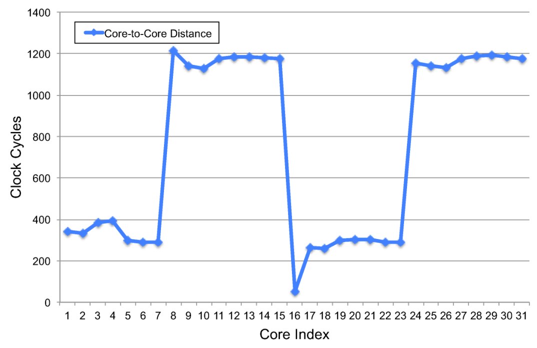


Fig. 4. Core Distances from Core 0 to other cores on Intel Xeon E5-2680.

C. Core Distances on Manycore Platforms

We see that core distance is not always correlating with the physical distance on chip. For a manycore system communicating over a ring network, such as the Xeon Phi™ 5110P coprocessor, one could expect the core distance as indicated with the green line in Fig. 5, where Core 0 has a shorter core distance to its neighbors than to the opposite core on the ring network. In contrast to the expectation, the measured core distances from Core 0 to Core *4n+1* *(0≤n≤59)* are shown in the blue line in Fig. 5. Except for Core 237, which is another hardware thread on the same physical core, Core 125 exhibits the shortest core distance to Core 0.

On a closer look, we note that the Intel Xeon Phi coprocessor uses a directory-based cache coherence protocol. The distributed tag directory maintaining cache coherence plays an important role in the core-to-core communication. Fig. 6 depicts the detailed communication model on this platform.

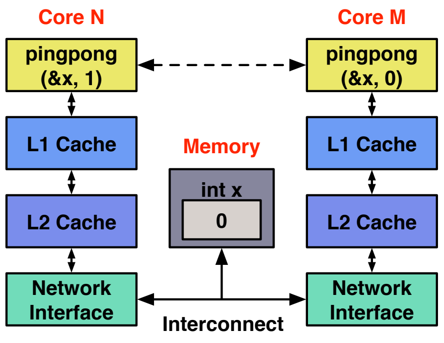


Fig. 2. Ping-pong communication for measuring core distance.

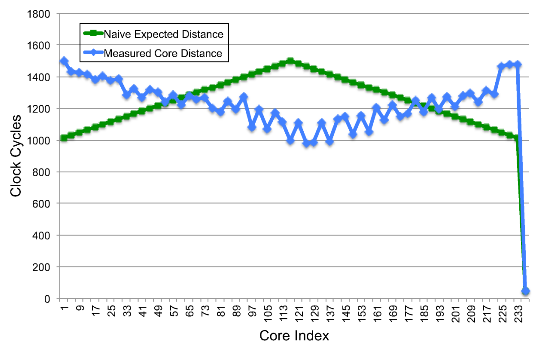


Fig. 5. Core distances from Core 0 to other cores on Xeon Phi 5110P.

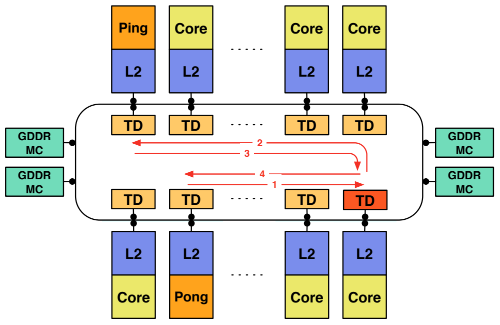


Fig. 6. Cache coherence via distributed tag directories (TD).

When one thread (**Pong** in Fig. 6) has a cache miss and needs to fetch an updated value, it first talks to the responsible tag directory (the red TD in Fig. 6) to find which core cache contains the new value (Step 1). Every memory reference the processor generates is mapped through a one-to-one hash function to a TD based on the physical address. Notably, the responsible TD is not necessarily co-located with the core generating the cache miss [2] and could be associated with any core on the chip. When the responsible TD finds another core (**Ping** in Fig. 6) owns the updated value, it sends a request for the new value to the specific core (Step 2), gets the value from that core (Step 3) and passes it back to the core exeperiencing the cache miss (Step 4). Finally, the TD updates the sharing status of the cache block in the first core.

D. Core Distances on Busy Manycore Platforms

When multiple pairs of communication happen concurrently, the core distances on the manycore processor become even more expensive and unpredictable. Fig. 7 shows the core distances from Core 0 to Core *4n+1* $(0 \le n \le 59)$ on a busy Xeon Phi coprocessor (green line), in comparison to the situation when only two cores communicate with each other on the chip (blue line, same as the blue line in Fig. 5). Here we run a Monte Carlo simulation which issues 60 pairs of concurrent communications and randomly distributes them on the ring network. As half of the chip is busy and the cores compete for the access to network, the core-to-core communication latency grows up to 10,000 cycles for one round trip. Also, the location of the responsible tag directory becomes negligible compared to the interference between parallel communications.

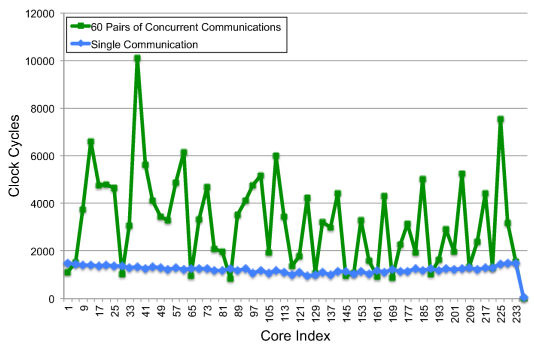


Fig. 7. Core distances from Core 0 to other cores at 50% core utilization.

III. Minimizing Inter-core Communication

In this section, we exploit the above observations and improve application performance by optimizing the thread-to-core mapping to minimize the on-chip communication overhead.

A. Mapping Communicating Threads Close to the TD

Based on the core distance curve in Fig. 5, we can infer the location of the responsible TD by finding the core index which has the shortest core distance, ignoring the logical cores on the same physical core. Algorithm 2 describes the function to find the responsible tag directory on the ring network. In Function **FindTD**, one thread is fixed to Core 0 and the other thread is set to the other cores on the ring network. Both threads run the **pingpong** function simultaneously. As one thread may start first and suspend for the other, we choose to use the core distance measured in the thread starting later (by checking timestamp T_1 of each thread). The function will find the minimum core distance and return the corresponding *coreindex* as the location of the identified tag directory.

Algorithm 2 Find the Responsible Tag Directory

```
1:  unsigned int FindTD (char $*var$)
2:  { for all $c \in$ ProcessorCores do
3:      set core affinity to 0 for Thread $th_1$
4:      set core affinity to $c$ for Thread $th_2$
5:      create Thread $th_1$ to run $t_1 :=$ pingpong($var, 0$)
6:      create Thread $th_2$ to run $t_2 :=$ pingpong($var, c$)
7:      if $th_2$ starts later then
8:        $coredist_{0,c} := t_2$
9:      else
10:       $coredist_{0,c} := t_1$
11:     end if
12:   end for
13:   return $coreindex$ where $coredist_{0,coreindex} = \min\{coredist_{0,c}\}$
14: }
```

With the knowledge of the TD location we can reduce the communication latency. By invoking the **FindTD** function at the beginning of the program, our approach profiles the application and determines the TD locations of the shared variables[1] in the application. Mapping threads close to these tag directories will reduce the onchip communication overhead[2]. Algorithm 3 and Algorithm 4 compare the thread initialization with and without optimization respectively. Rather than using the default thread attributes in Algorithm 3, Algorithm 4 first finds the tag directories td_i of each communication channel ch_i, and then sets the core affinity of the communicating thread close to td_i. **GetCore** returns a core index which is near td_i and tries to balance work load among all available cores in a greedy fashion. Next, all communicating threads are created with the affinity-optimized thread attributes.

Fig. 8 shows the optimized core distances for the Intel Xeon Phi coprocessor, with one thread placed at the responsible TD (Core 125, using the same TD as Fig. 5). The minimum core distance here is about 500 cycles, when mapping the other thread next to the tag directory. This is much lower than the core distance in Fig. 5 which is as high as 1500 cycles. Also,

[1]The size of a shared variable should be smaller than one cache block (64 bytes on Intel Xeon Phi) and mapped to one tag directory.

[2]As the hash function is unknown and the mapping is based on the physical address of the memory reference, it is very difficult to allocate a shared variable whose responsible tag directory is close to the communicating cores. So instead of moving the TD, we move the threads.

Algorithm 3 Thread Initialization without Optimization

```
1: for all Thread $th_{i,1}, th_{i,2}$ using Channel $ch_i$ do
2:     $th_{i,1}$.ThreadCreate($DefaultThreadAttributes$)
3:     $th_{i,2}$.ThreadCreate($DefaultThreadAttributes$)
4: end for
```

Algorithm 4 Thread Initialization with Optimization

```
1: for all Channel $ch_i$ do
2:     $td_i$ := FindTD(addressof($ch_i$))
3:     set core affinity to GetCore($td_i$) in $ThreadAttributes_{i,1}$
4:     set core affinity to GetCore($td_i$) in $ThreadAttributes_{i,2}$
5: end for
6: for all Thread $th_{i,1}, th_{i,2}$ using Channel $ch_i$ do
7:     $th_{i,1}$.ThreadCreate($ThreadAttributes_{i,1}$)
8:     $th_{i,2}$.ThreadCreate($ThreadAttributes_{i,2}$)
9: end for
```

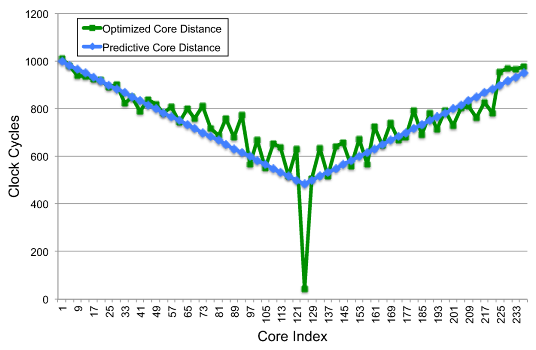


Fig. 8. Optimized inter-core communication latency.

the core distance is now *predictable* with the physical location of the core, as indicated by the blue line in Fig. 8. The further two cores are on the ring network, the larger the core distance is between them.

B. Pizza Slice Algorithm

In a busy situation where many pairs of core-to-core communication run simultaneously, transfer latency is extremely expensive (Fig. 7) and unpredictable. Here, in order to reduce the interference and improve onchip communication, we propose a software algorithm which divides the ring network into a given number of sections and localizes the communication into these sections. We name this the *Pizza Slice Algorithm* due to its similarity with a sliced pizza (Fig. 9). By evenly distributing the concurrent communications onto a given number of different slices of the ring network, our approach maps threads onto cores of one section and selects a shared variable whose responsible tag directory is in the same section[3].

Fig. 10 compares the core distances from Core 0 to Core *n* $(1 \leq n \leq 39)$ in one section, before and after applying the *Pizza Slice Algorithm* to the Xeon Phi coprocessor in which half of the chip is busy communicating. Core distances are measured using Monte Carlo simulation in both cases. With the *Pizza Slice Algorithm*, the threads and tag directories are localized into one section, and communication latencies decrease dramatically (from more than 7000 to less than 1000 cycles).

[3]We need to allocate a few (about 25%) more shared variables than defined in the program so as to find enough communication channels in each section.

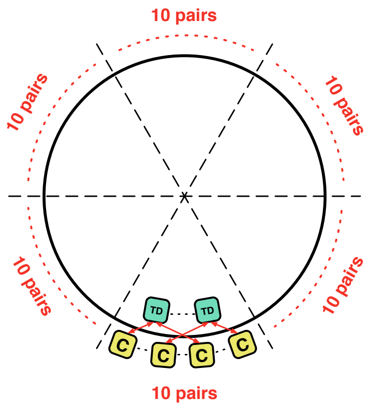


Fig. 9. "Pizza Slice Algorithm" for cores and TDs on a ring network.

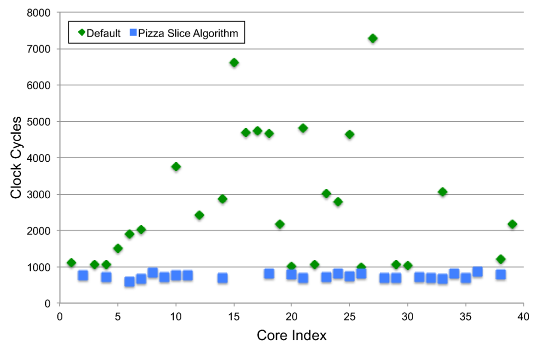


Fig. 10. Comparison of communication latency before and after applying the Pizza Slice Algorithm.

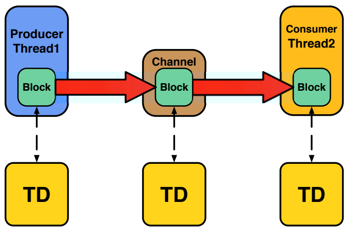
IV. Experimental Results

To demonstrate the performance speedup for actual applications, we present experimental results of two communication-intensive benchmarks: a producer-consumer example and a Mandelbrot set renderer model. All the experiments are conducted on the Intel Xeon Phi 5110P coprocessor running at 1.052 GHz.

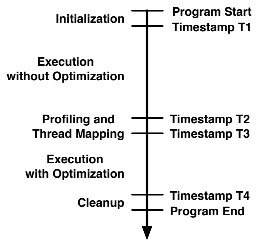
A. Producer-Consumer Model

Our producer-consumer model is a classic example of a multiprocess synchronization problem. Fig. 11(a) shows the block diagram where Producer and Consumer are children of the root thread, communicating a data value through a spin-locked channel which has a buffer size of one. The Producer generates a data value and puts it into the channel buffer. After the Consumer fetches the data, the Producer resumes to generate new value. In addition to the channel buffer, the Producer and Consumer respectively own a local variable to store the communicated data. The channel buffer is accessed by both Producer and Consumer threads, which contain a copy of the data block in their local caches. Thus, the responsible tag directory for the channel data is of major concern in the communication. To speed up the synchronization, we can map Producer and Consumer threads close to the tag directory of the channel block.

To ensure a fair comparison, we run both the default and our optimized version in the same process, so that the same TDs are used. Fig. 11(b) shows the timeline of our experimental evaluation. First, the model runs with Linux default



(a) Block diagram.


Initialization
Program Start Timestamp T1
Execution without Optimization
Profiling and Thread Mapping
Timestamp T2
Timestamp T3
Execution with Optimization
Cleanup
Timestamp T4
Program End

(b) Execution timeline.

Fig. 11. Producer-consumer example.

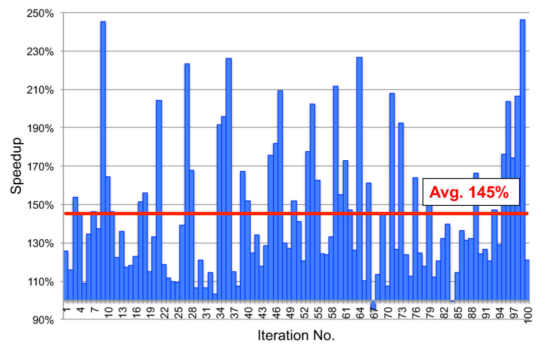


Fig. 13. Speedup of the producer-consumer example when mapping threads close to the TD.

settings (from *T1* to *T2*). Next, we profile the application and apply the *Pizza Slice Algorithm*. The overhead of profiling and optimizing the thread mapping is measured by *T3-T2*. Finally, the model runs again with the optimized thread-to-core mapping from *T3* to *T4*. The performance speedup of the model is calculated by dividing *T2-T1* by *T4-T2* (i.e. including the profiling overhead).

1) Assigning Threads onto One Physical Core: According to Fig. 5, communication between logical threads on the same physical core is the fastest possible option. Fig. 12 shows the performance improvement of the producer-consumer model by mapping both threads to the same physical core. Running the benchmark for 100 iterations, we show the statistical results of the application speedup.

As the inner core distance is 10 times smaller than that between physical cores, the optimized producer-consumer model achieves an order of magnitude higher execution speed (7x to 16x) than the original model. The variation of the performance speedup is due to the varied locations of the responsible TD and resulting execution time in the original model. On average the performance increases by 11.66x.

2) Assigning Threads to Close Cores: In most applications, it is a bad idea to map threads onto the same physical core. Fig. 13 shows the performance of the producer-consumer example when assigning the threads onto cores close to the TD (one core co-located with the tag directory, and the other next to it). Here, our approach shortens communication latency and gains speedup in most experiments, up to 2.5x. In some rare cases (2 of the 100 iterations), if the Linux default scheduler ”luckily” sets threads to the optimal cores, our approach performs only a little worse (95%) due to our overhead. The mean value of the speedup is 1.45x.

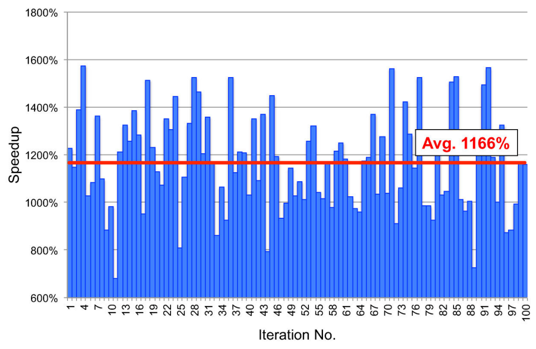


Fig. 12. Speedup of the producer-consumer example when mapping threads onto the same physical core.

B. Mandelbrot Set Renderer Example

As an example of highly parallel graphics applications, we use a renderer for Mandelbrot set images [4]. Fig. 14 shows its block diagram. In the model, there are four Coordinators which work on four separate slices of each frame and send coordinates in the complex plane to 8 Mandelbrot worker threads. Each worker thread calculates whether the point belongs to the Mandelbrot set or not. Zooming into an interesting area of the image, our application generates a series of 100 Mandelbrot set images.

Fig. 15(a) shows that our *Pizza Slice Algorithm* accelerates the graphics application by a maximum speedup of 150% and a minimum of 95%. The Pizza approach optimizes the application by mapping each Coordinator and its associated worker threads and channel tag directories into one of four sections on the ring network. Worker threads are assigned to the same cores of the responsible TDs (or next to TD to balance CPU load) and the Coordinator is placed in the middle of the section. Fig. 15(b) shows the overhead of profiling and optimization (time *T3-T2* in Fig. 11(b)). Compared to the execution time of the new model (*T4-T2*), the profiling overhead is less than 0.8 second absolute, or 1% of the execution time. The speedup is 125% on average, as shown in the histogram in Fig. 15(c).

V. Related Work

With the fast increasing number of processing cores on a single die, we expect hundreds and even thousands of cores integrated on the Chip Multiprocessor (CMP) systems, which are known as manycore platforms. Currently, some manycore

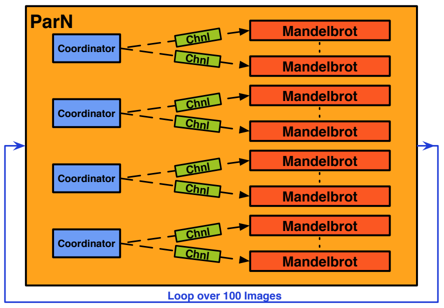
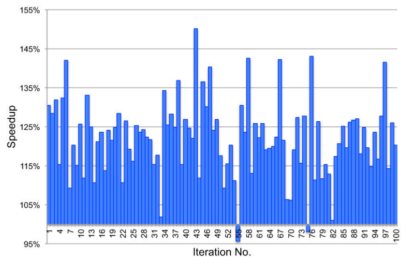
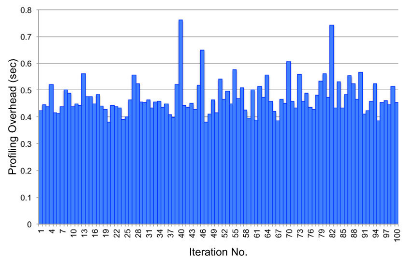


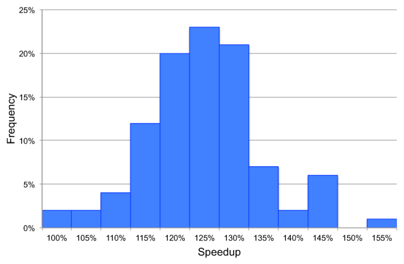
Fig. 14. Block diagram of the Mandelbrot renderer example.



(a) Performance speedup.



(b) Profiling overhead.



(c) Histogram of the speedup.

Fig. 15. Apply Pizza Slice Algorithm to the Mandelbrot renderer.

systems are readily available, e.g. Tilera [5], Intel's TeraFlop [6] and MIC [1]. Together with GPGPU and distributed shared memory (DSM) systems, these high-performance computing platforms provide abundant parallel processing power. How to efficiently exploit these platforms has been a hot topic in recent researches. [7] and [8] address the thread mapping strategy in heterogeneous multiprocessor systems by utilizing dynamic thread-to-core assignment. [9] proposes a formal model to characterize threads and cache hierarchy of GPGPUs and generate an optimized thread mapping scheme. [10] also optimizes the shared cache GPGPU, but targets applications with irregular data accesses. [11] and [12] describe novel approaches to schedule threads on DSM systems, taking memory traces and hierarchy into consideration. [13] addresses the problem of application mapping on a Network-on-Chip (NoC) multiprocessor. In [14] and [15], the authors propose an analytical model to characterize programs, machines and costs for multiprocessor platforms with hierarchical memory architectures.

In comparison to these works, our approach differs in two aspects. First, we are optimizing thread mapping on homogeneous manycore platforms with distributed tag directories maintaining cache coherence. As this type of platform is a different processor architecture, the problem of thread-to-core mapping cannot be addressed by the existing methods for CMPs with hierarchical memory system or GPGPUs. Second, we propose the measurement of core distance to quantify the core-to-core transfer latency precisely, rather than using theoretical values as in other work.

VI. Conclusion

In this paper, we propose a software approach to optimize the thread-to-core mapping for homogeneous manycore processors with distributed tag directories. By profiling the application and optimizing thread assignments, our approach can reduce the core-to-core transfer latency significantly and improve the application performance by more than 25% over the Linux default strategy.

In future work, we plan to extend our approach to more applications and platforms. Currently, we only consider communication where the shared variables are within one cache block and mapped to one TD. Based on the further refinement and measurement of core distances, we intend to model more general applications and platforms, and automatically generate optimized thread-to-core mapping.

Acknowledgment

The authors thank Intel Corporation for the access to a Xeon Phi™ Coprocessor and for funding this research.

References

[1] Intel Corporation, *Intel® Xeon Phi™ Coprocessor Datasheet*, June 2013.

[2] Intel Corporation, *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*, April 2013.

[3] J. Jeffers and J. Reinders, *Intel® Xeon Phi™ Coprocssor High-Performance Programming*, Waltham, USA: Morgan Kaufmann, 2013.

[4] B. Mandelbrot, "Fractal aspects of the iteration of $z- > \lambda z(1-z)$ for complex λ and z", *Annuals of the New York Academy of Sciences*, 1980

[5] Tilera Corporation, *Tilera multicore processors*, http://www.tilera.com /products/processors.

[6] S. Vangal, J. Howard, G. Ruhl, and e.t.c., "An 80-tile sub-100-w teraflops processor in 65-nm cmos", *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29-41, Jan 2008.

[7] G. Liu, J. Park, D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems", in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Asheville, NC, Oct 2013.

[8] M. Becchi, P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures", in *Proceedings of the 3rd conference on Computing frontiers (CF)*, New York, USA, 2006.

[9] B. Lai, H. Kuo, J. Jou, "A Cache Hierarchy Aware Thread Mapping Methodology for GPGPUs", *IEEE Transactions on Computers*, Feb 2014.

[10] H. Kuo, K. Chen, B. Lai, J. Jou, "Thread affinity mapping for irregular data access on shared Cache GPGPU", in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, Sydney, NSW, Jan 2012.

[11] K. Thitikamol, P. Keleher, "Thread migration and communication minimization in DSM systems", in *Proceedings of the IEEE*, Mar 1999.

[12] F. Song, S. Moore, J. Dongarra, "Feedback-directed thread scheduling with memory considerations", in *Proceedings of the 16th international symposium on High performance distributed computing (HPDC)*, New York, USA, 2007.

[13] G. Chen, F. Li, S. Son, M. Kandemir, "Application mapping for chip multiprocessors", in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, Anaheim, CA, June 2008.

[14] F. Song, S. Moore, J. Dongarra, "Analytical modeling and optimization for affinity based thread scheduling on multicore systems", in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, New Orleans, LA, Aug 2009.

[15] E. Molina da Cruz, M. Zanata Alves, A. Carissimi, and e.t.c., "Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms", in *Parallel and Distributed Processing Workshops and Phd Forum (ISDPSW), 2011 IEEE International Symposium on*, Shanghai, May 2011.