Experience on Optimizing Irregular Computation for Memory Hierarchy in Manycore Architecture *

Guangming Tan, Dongrui Fan, Junchao Zhang

Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Science.

tgm@ncic.ac.cn, {fandr,jczhang}@ict.ac.cn

Andrew Russo, Guang R. Gao

Computer Architecture & Paralel Systems Laboratory, Electrical & Computer Engineering, University of Delaware.

{russo, gao}@capsl.udel.edu

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming-Parallel Programming

General Terms Experimentation, Performance

Keywords percolation, memory hierarchy, synchronization, irregular computation.

1. Introduction

With the rapid advance of multi-core/many-core chip technology, we have recently witnessed many proposals from both the industry and academia which actively exploit the design space of many-core chip architecture. While the number of core is increasing, the memory hierarchy design and architectural support of communication or synchronization are becoming controversial problems for performance and programming. The motivation of behind this work is to be able to identify not only how programmers will use the mechanisms provided in the emerging many-core architecture, but the relative usefulness of various mechanisms as evidenced by their impact on application performance. This paper focuses on addressing the issues of locality for an irregular application – graph traversal – on two novel many-core chip architectures – IBM Cyclops64 (C64) [1] and ICT GodsonT [2], which are manycore architectures designed to serve as a dedicated petaflop compute engine.

A C64 chip (see figure 1) has 80 processors, each with two thread units, a floating-point unit and two SRAM memory banks of 32KB each. The C64 chip has no data cache and features a three-level (Scratchpad (SP) memory, on-chip SRAM, off-chip DRAM) memory hierarchy (see figure 2). On-chip resources are connected to a 96-port crossbar network, which sustains all the intra-chip traffic communication and provides access to the routing ports that connect each C64 chip to its nearest neighbors in the 3D-mesh network. Each processor is connected to a crossbar network that can deliver 4GB/s per port, totaling 396GB/s in each direction.

A GodsonT chip has 96 processors connected by two level hierarchy on-chip networks (Figure 3). All processors are grouped into 5×5 tile nodes shaped as 2-D mesh interconnection. Except for one synchronization node, each one of other 24 tile nodes consists of 4 processors. Within a tile node, one 7*7 full crossbar providing non-blocking communication connects 4 processors, L1 cache subsystem (I-cache and D-cache) and the router interface. The cache hierarchy is composed of "private" L1 cache (write-through)shared by 4 processor in the same tile node and global L2 cache shared by all processors.

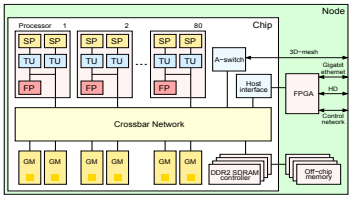

Figure 1. C64 chip architecture

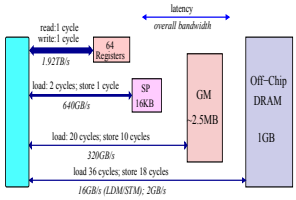

Figure 2. Memory hierarchy on C64

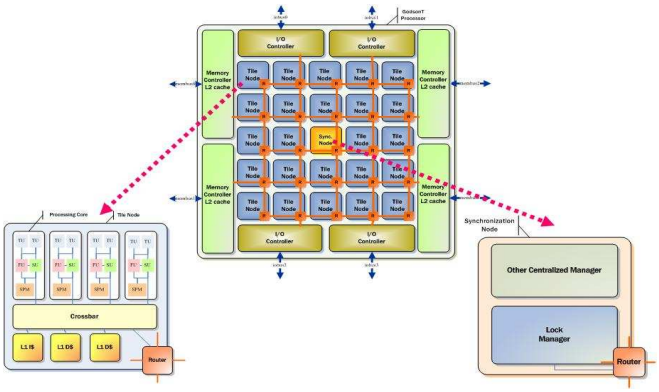

Figure 3. GodsonT chip architecture.

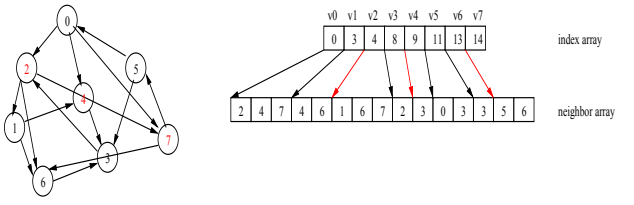

Figure 4. Adjacency array of a graph.

We illustrate the experience of programming many-core chip architecture on an important irregular application: computing betweenness centrality (BC) [3]. BC algorithm is composed of breath-

* This work is partially supported through the support from National Science Foundation (CNS-0509332); DE-FC02-01ER25503; NSFC (60633040) and (60736012); National 973 Fundamental Research (2005CB321600).

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

first search (BFS) phase and backtrace phase. Since the memory access behavior is similar between these two consecutive phases, we mainly present the BFS phase as an example to describe our optimization strategies. An space efficient data structure of scale free sparse graph is adjacent array (Figure 4). During the BFS phase, a queue is used to maintain the current vertices that is being extended (visiting the neighboring vertices of a vertex is referred to as an extension operation). In a scale free sparse graph, the degrees or neighbors of vertices are highly variable. Assume that the nodes v_2, v_4 and v_7 in Figure 4 are currently in queue from which we pick nodes and process them. Notice that not only the neighboring nodes of v_2, v_4 and v_7 are located in different region in the adjacency array, but also the strides between the different regions are not constant. Due to random distribution of neighbor vertices, access to other arrays like recording distance and path is descrete. Such non-contiguous or non-linear memory access pattern cannot benefit from traditional prefetching or speculation.

2. Percolation in Explicit Memory

To exploit dynamic locality, we made one key observation when understanding the locality behavior of irregular applications. There are "non-linear locality" that exists within these applications, and unfortunately those non-linear locality are spread out across dynamic envelope of the application state. Since most architecture (including many-core architectures) are designed to exploit "linear locality" it is important transform (using *percolation programming*) non-linear locality into linear locality just in time for the computation. For instance, consider the example shown in Figure 4, and assume that v_2, v_4 and v_7 are currently in queue, and assume we extend node v_2, to bring in (percolate in) nodes v_1, v_6 and v_7. Since these three nodes are contiguous we exploit the locality among them and arrange their neighbor vertices in a linear contiguous memory (within the core memory). There are three pipelined parts to percolation programming: (1) Collect data from non-contiguous location just in time to obtain just in locale in the on-chip memory for the computation phase; (2) Compute the relevant information based on just in locale on-chip data; and (3) Finally, map the information thus computed back to off-chip memory. The first and last phase form the memory task and are mapped to helper threads by the runtime system. The second phase is the main computation phase that is mapped to several computation threads.

3. Lock-based Consistency in Cache Memory

While the scalability wall of traditional snoopy- or directory-based cache coherence on a large scale many-core is well-known, irregular programs like BC algorithm make it worse. First, the data structure in a BC-like graph traversal program is un-partitionalbe. For example, the adjacent array and other arrays recording path have to be shared by all threads because the memory accesses to these arrays are dynamically determined at runtime. Second, there exist lots of producer-consumer (P/C) patterns. In BFS phase, the access to arrays recording path depends on the neighbor vertices. If a neighbor vertex has been visited before, it would not been counted in the current extension. However, since the distribution of neighbor vertices in a scale-free graph is almost random, every extension operation in different threads requires a synchronization of their "private" data. Therefore, a traditional coherence protocol like MESI results in amount of cache misses and network traffic.

We proposed a memory consistency model lies between release consistency and scope consistency [5] in GodsonT. Note that a specific tile node is designed for synchronization, we implement hardware lock based cache consistency protocol, which is conceptually similar to those lock based protocols in software shared memory systems, between private L1 caches and shared L2 cache.

```
P1:                          P2:
Producing data;              ...
BARRIER;                     BARRIER;
...                          ACQUIRE(L);
                             Consuming data;
                             RELEASE(L);
```

Figure 5. P/C Synchronization on GodsonT cache consistency

Table 1. The comparison of time (seconds) on three platforms.

#threads	C64	GodsonT	Niagara
4	2.875	4.056	2.540
8	1.521	2.734	3.125
16	0.861	2.074	5.126
32	0.549	1.504	10.539

Two hardware synchronization primitives – Acquire and Release – which have both locking and memory ordering semantics are implemented for programming in GodsonT. It is the programmer's responsibility to define coherence scopes with Acquire/Release pair. Within a coherence scope, all modified memory locations are propagated two all other threads that share the same scope. Another important observation is that the L1 cache is write-through, programmers (or compiler) only needs to mark the scope of consistency with Acquire/Release pair in the consumer codes. Figure 5 shows an example of P/C synchronization.

4. Experimental Results

We implemented three parallel programs of BC algorithm according to HPCS SSCA2 [4] benchmark on C64, GodsonT and Sun Niagara. The program on Sun Niagara is a highly optimized OpenMP implementation. Both C64 and GodsonT support a specific multithreading virtual machine providing a Pthread-like programming style. Although the number of processing cores in systems will grow rapidly, the computational power of individual processing units is likely to be reduced. This trend leads a shift of measuring speedup from *weak scaling* to *strong scaling*, where speed is achieved when the number of processors increased while the overall problem size is kept constant. Thus, our experiments focus on the performance of small problem size.

Table 1 reports the execution time for a graph with $n = 2^{10}$ vertices and $8n$ edges. Percolation for creating locality just in time achieves sub-linear speedup on C64 with software-managed memory hierarchy. Lock-based consistency for managing irregular producer-consumer coherence also obtains reasonable scalability on GodsonT with hardware-managed memory hierarchy. The experiment results implicates that an important key to make many-core chip architecture successful is the software technology.

References

[1] Denneau, M., and Warren, Jr., H. S. 2005. 64-bit Cyclops: Principles of operation. *IBM Tech-report.*

[2] D. Fan, N.Yuan, J. Zhang. 2007 Design Philosophy and Microarchitecture of Many-core GodsonT. *ICT Tech-report.*

[3] Brandes, U. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25*, 2, 163–177.

[4] Bader, D. A. Hpcs scalable synthetic compact applications 2 graph analysis. *www.highproductivity.org/SSCABmks.htm.*

[5] L. Lftode, J. P. Singh, K. Li. 1996. Scope Consistency: A Bridge between Release Consistency and Entry Consistency *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*