Template for Preparation of Manuscripts for *Tsinghua Science and Technology*

This template is to be used for preparing manuscripts for submission to *Tsinghua Science and Technology*. Use of this template will save time in the review and production processes and will expedite publication. However, use of the template is not a requirement of submission. Do not modify the template in any way (delete spaces, modify font size/line height, etc.).

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 0?/?? pp???-???
Volume 18, Number 3, June 2013

Exploitation of Locality for Energy Efficiency for Breadth First Search in Fine-grain Execution Models

Chen Chen*, Souad Koliai, and Guang R. Gao

Abstract: In the upcoming exa-scale era, the exploitation of data locality in parallel programs is very important because it benefits both program performance and energy efficiency. However, this is a hard topic for graph algorithms such as the breadth first search (BFS) due to the irregular data access patterns.

This study analyzes the exploitation of data locality in the BFS and its impact on the energy efficiency with the Codelet fine-grain dataflow-inspired execution model. The Codelet Model more efficiently exploits data locality than the OpenMP-like execution models which traditionally focus on coarse-grain parallelism inside loops. A BFS algorithm is then given to exploit the locality between two loop iterations that belong to two different loops (inter-loop locality). This kind of locality can be exploited by the codelet model but not by traditional coarse-grain execution models like OpenMP.

Tests were performed on *fsim* which is a simulation platform developed by Intel for the UHPC project to design future exa-scale architectures. The results show that this BFS algorithm saves up to 7% of the dynamic energy for memory accesses compared to a BFS implementation based on OpenMP loop scheduling.

Key words: Breadth First Search; Locality; Fine Grain; Execution Model.

1 Introduction

In the upcoming exa-scale era, architecture design will have many cores on a chip, with many chips forming a system. The energy efficiency then becomes very important because of the large power consumption [1, 2]. Hardware caches increase energy use due to unnecessary memory accesses (loading more data than needed) and false sharing. In some new architecture designs, a chip has several levels of globally shared memory for data transfers among cores. Moreover, each core has local storage that can be accessed faster with lower power consumption compared to accessing shared memory. The local storage buffers data that might be used in the near future, thus replacing caches. Some examples of such architectures are the IBM CELL Broadband Engine[3], IBM Cyclops64 [4], and Intel UHPC Straw-man architecture [5].

In architectures with local storage attached to each core, parallel programs require software and programmer efforts to analyze and decide how to efficiently utilize the local storage via exploitation of locality. This is extremely important for graph applications such as the breadth first search (BFS) because such applications are normally memory access intensive, and have irregular data access patterns that complicate exploiting locality.

Currently, there are only a few studies of the energy efficiency issue of the BFS problem. Satish *et al.* [6] claimed that their work to be "the first paper showing energy efficiency on Graph500* benchmark". They

• The authors are with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, 19711, USA. {chenchen, koliai, ggao}@capsl.udel.edu

* To whom correspondence should be addressed.

Manuscript received: year-month-day; revised: year-month-day; accepted: year-month-day

*The Graph500 [7] is an effort to establishe a set of large-scale graph benchmarks for high performance computing related applications.

applied their BFS optimization on an Intel architecture with 3-level caches. The most recent work on an architecture with local storage appears to be the work of Scarpazza *et al.* [8] on the Cell Broadband Engine. Therefore, more work is needed to analyze energy efficiency of BFS on architectures with local storage.

The execution of a typical parallel BFS algorithm consists of many steps. Each step executes a parallel *for* loop to explore part of the BFS tree. There are then synchronizations (e.g., global barriers) between every two adjacent loops. The BFS algorithm may have two kinds of locality. One is intra-loop locality between loop iterations in the same loop. The other is inter-loop locality between loop iterations in different loops. With this observation, this paper makes the following contributions:

- Intra-loop locality is easy to exploit but inter-loop locality is hard to exploit in OpenMP-like execution models which traditionally focus on coarse-grain parallelism inside loops. Fine-grain execution models can easily exploit both types of locality.

- A BFS algorithm is given to exploit inter-loop locality based on the Codelet Model [9] which is a fine-grain dataflow-inspired execution model.

- The localities and energy efficiencies of the Codelet Model and the OpenMP-like execution model are compared.

The BFS algorithms were implemented on *fsim* which is a simulation platform developed by Intel for the UHPC project [10] to design future extreme-scale architectures. The results show that the BFS algorithm reduces the memory access dynamic energy consumption by 7% compared to the BFS implementation based on OpenMP loop scheduling.

2 Background

This paper shows how to exploit locality and save energy for the BFS on architectures with local storage via the Codelet fine-grain execution model. This model can be mapped to the Intel UHPC straw-man architecture for the design of future exa-scale architectures. This section introduces the Codelet Model, the architecture, and the BFS algorithm.

2.1 Codelet Model

The Codelet Model is a fine-grain dataflow-inspired computational model that relies on the dataflow paradigm to exploit parallelism on future many-core architectures. The units in computation of the Codelet Model are called *codelets*. Each codelet is a piece of sequential code that can be executed without interruption. Once a codelet starts execution, it does not need to wait for any synchronization. The model relies on explicit data dependence specified between the codelets. The codelets and their dependencies form a directed graph called the codelet graph. During execution, a codelet runtime maintains the dependencies and schedules the codelets to the available cores. The codelet runtime ensures the dependencies via signals of fulfilled events from one codelet to the specific codelet that is waiting for the events.

The Codelet Model is presented in the context of a parallel abstract machine model. The abstract machine consists of many nodes connected together via an interconnection network. Each node contains a many-core chip. The chip may have 1,000 to 2,000 processing cores organized into groups (clusters) linked together via a chip interconnect. These cores will be quite simple and take less transistor space. Compared to large cores, these smaller simpler cores consume less power and are more simply packed on a single die. The cores in this abstract machine model are grouped hierarchically. The grouping of cores promotes locality in applications, since tasks can also be grouped to target a specific hierarchical level in the machine.

In an abstract machine, each group contains a collection of computing units (CU) and a scheduling unit (SU). By diversifying cores, natural strengths are given to different components in the architecture to perform different tasks. A SU is responsible for runtime operations and steering computations. The number of SUs needed differs from the number of CUs. A heterogeneous approach maps a reasonable amount of CUs to the SUs to provide the optimal amount of workers to the schedulers. This division of labor (scheduling to computation) and ratio (CUs to SUs) should lead to a more power efficient architecture.

An abstract view of the computation unit is shown in Fig. 1 with a group of computation cores and some local memory. A node may also contain other resources, most notably additional memory which will likely be DRAM or other external storage.

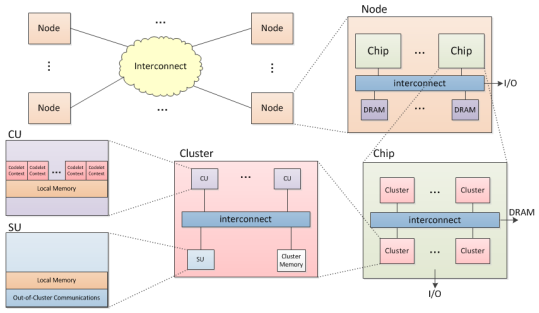

Fig. 1 **Codelet Abstract Machine Model**

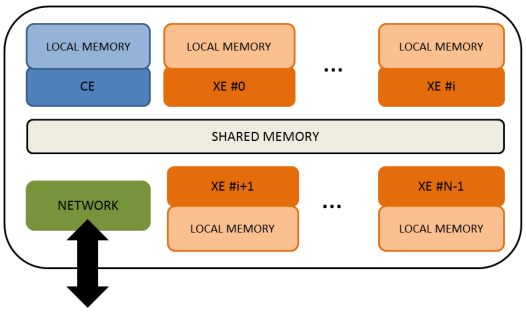

Fig. 2 **A *fsim* shared memory *block* with N XEs.**

The features of the abstract machine model are:

- **Hierarchical**. A system will contain many chipsi and each single chip will contain many different level of processors:
  - Computations: nodes, chips, clusters with computation units (CU) and synchronization units (SU).
  - Memory: main (shared) memory and local memory (attached to a given computation unit).
- **Heterogeneous**. Elements have different roles such as:
  - Computation units (the most numerous type of cores) handle the computations.
  - Scheduling units handle exceptions, hardware failures, out-of-cluster requests, etc. Specifically, they handle any memory request that goes outside the cluster, or is received from an out-of-cluster location.

2.2 Architecture

The *fsim* simulation platform was used to test different implementations of the BFS algorithm. *fsim* was developed to simulate (in software) the current Intel hardware architecture prototype in the UHPC project to test future exa-scale architectures. This simulated architecture include two types of processors [5]:

- **Control engines (CEs):** cores which execute instructions in the distributed runtime environment, including support for peripherals, but not direct user code. A CE matches to a synchronization unit in the codelet abstract machine model.
- **Execution engines (XEs):** simple, plentiful, very low-power cores optimized for HPC applications that may be heterogeneous when disparate types of fixed-function logic or accelerators are useful. The XEs run the application (user) code. Each XE is assigned thread-local storage such as registers and private scratchpad memories. An XE matches to a computation unit in the codelet abstract machine model.

fsim simulates the Intel's straw-man architecture. This architecture is composed of different blocks of XEs and CEs in the processor chip:

- **Block:** A group of N XEs and 1 CE with a local memory for each engine and a shared memory between all engines.
- **Cluster:** A group of *blocks* connected by a specific interconnect and sharing another level of memory.
- **Chip:** A group of *clusters* connected by a crossbar switch and sharing memory.

Fig. 2 illustrates one *block* of N XEs. The tests used a block of 8 XEs.

2.3 Breadth First Search

The basic breadth first search algorithm used in this study comes from Graph500 [7]. Graph500 establishes a set of large-scale graph benchmarks for high performance computing related applications.

The pseudo code of the breadth first search kernel is shown in Algorithm 1. The algorithm starts with a root vertex in the search list. The algorithm finds its neighbors each vertex in the search list and puts them into a new search list for future searches. Each vertex is marked before it is put into the search list. In this way, the algorithm avoids repeated searches of the same

Algorithm 1: **BFS algorithm pseudo code**

input : Undirected graph G and starting vertex $root$
output: A BFS tree represented by a vector $parent$ that stores the parent of each vertex in the BFS tree
Data: Q is the current search list and Q' is the search list for the next turn

PSEUDO CODE:

```
foreach element e of parent do e ← −1;
parent[root] ← root;
Q←{root};
while Q ≠ ∅ do
    Q' ← ∅;
    foreach v ∈ Q in parallel do
        foreach v' adjacent to v in G do
            if parent[v'] == −1 then
                parent[v']←v;
                Q'←Q' + {v'};
    synchronization;
    Q ← Q';
```

vertex. The algorithm terminates when the search list is empty.

3 Algorithm

This section describes how to exploit locality for the BFS for both the coarse-grain execution model and the fine-grain execution model. The BFS algorithm in Algorithm 1 has two kinds of locality. One is intra-loop locality between two loop iterations in the same parallel *for* loop. The second is inter-loop locality between two loop iterations in different loops. Section 3.1 presents an example to explain these two kinds of locality. Then Section 3.2 and 3.3 introduce how to exploit the two kinds of locality in the BFS algorithms.

3.1 Motivation

This section provides an motivating example that explains how the codelet model can exploit both intra-loop locality and inter-loop locality. As explained in Section 1 and Section 2.3, execution of the BFS algorithm unfolds into many parallel *for* loops that are interleaved by synchronization between every two adjacent loops. The program execution can be easily represented as a codelet graph.

Fig. 3 shows an example of a piece of the program execution with two parallel *for* loops where each loop has 4 loop iterations. Each loop iteration, as well as the start and the end of each loop, are all represented as codelets. The dependencies among the codelets are obvious: (1) Each loop iteration depends on the starting

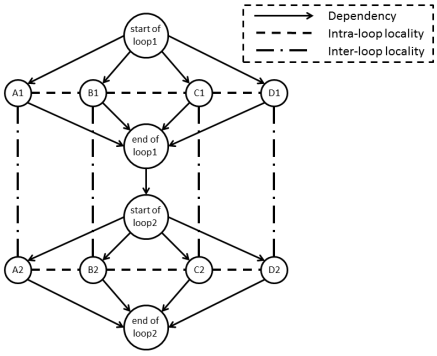

Fig. 3 **An example of intra-loop and inter-loop localities**

codelet of its loop. (2) The ending codelet of the loop depends on all the loop iterations in the same loop. (3) The starting codelet of the second loop depends on the ending codelet of the first loop to indicate the synchronization. The codelet graph has the two kinds of locality:

- **inter-loop locality:** In the codelet graph, the codelets for the loop iterations in the same loop may have spatial locality if their memory accesses are continuous. For example, if codelets $A1$ and $B1$ access continuous memory, they may be scheduled to the same processor. Then a multi-word load instruction may load that continuous memory, which is more energy efficient than multiple single-word load instructions.

- **inter-loop locality:** Fig. 3 shows that the codelets for the loop iterations in different loops may have temporal locality if the data produced by one codelet will be consumed by another codelet. For example, if codelet $A1$ produces some data that will be consumed by codelet $A2$, they may be scheduled to the same processor so that $A1$ may store the produced data locally (e.g., in registers or local storage) for future use by $A2$.

Coarse-grain execution models such as the OpenMP model easily exploit intra-loop locality because the locality is within one loop. For example, the OpenMP model allows static and dynamic scheduling with user specified chunk sizes to exploit intra-loop locality.

However, an OpenMP-like model cannot easily exploit inter-loop locality because the locality may cross several loops. Such a case needs fine-grain execution models. For example, the codelet model may

schedule $A1$ and $A2$ to the same core. Then, $A1$ may store the produced data in the local storage of the core for use by $A2$.

3.2 BFS Algorithm in the Coarse-Grain Model

In Algorithm 1, adjacent loop iterations in the same loop have spacial locality because they access continuous elements in Q. Therefore, executing the loop iterations in chunks will save energy. There are fewer accesses on the global shared queue, which reduces the total number of global memory accesses. Moreover, an energy efficient multi-word load instruction can be used to replace multiple single-word load instructions.

The BFS algorithm has irregular memory access pattern. So workload balancing will also impact the energy efficiency by reducing the idle time of the threads. Therefore, the algorithm implements both static scheduling and dynamic scheduling policies based on the OpenMP standard. Both scheduling polices spawn a number of threads (equal to the number of cores on the chip) at the beginning of the program. Then, one thread (master thread) initializes the data structures that are shared by all the threads. The parallel execution starts when the work is distributed among all the threads. After that, each thread will chose its working data based on the scheduling policy. The static and dynamic scheduling policies can be explained as follows:

- **Static Scheduling:** The static scheduling policy follows the OpenMP standard. The policy partitions the iteration space of each parallel *for* loop into chunks. Then it distributes the chunks to the spawned threads in a Round Robin fashion. Each thread needs to wait at the synchronization point when it completes the workload in its chunk. Static scheduling is usually used for better exploitation of data locality. However, the unbalanced nature of the BFS algorithm results in many idle threads with small workloads that complete their work much earlier than their siblings.

- **Dynamic Scheduling:** The dynamic scheduling follows the OpenMP standard as well. Each thread takes a chunk of work at beginning. Once a thread completes its work, it tries to take a chunk of new work from the shared work queue (the search list Q in the algorithm). Dynamic scheduling is a natural choice for BFS due to its irregular nature. This reduces the idle time of each thread, and thus reduces the energy waste.

Algorithm 2: **BFS algorithm pseudo code for the coarse-grain execution model. The static and dynamic scheduling use the same algorithm except that the chunk assignments are static or dynamic.**

input : Undirected graph G and starting vertex $root$
output: A BFS tree represented by a vector $parent$ that stores the parent of each vertex in the BFS tree
Data: Q is the current search list and Q' is the search list for the next turn

PSEUDO CODE:

```
foreach element e of parent do e ← −1;
parent[root] ← root;
Q←{root};
while Q ≠ ∅ do
    Q' ← ∅;
    foreach chunk C of vertices ⊂ Q in parallel do
        load C into local memory C';
        foreach v ∈ C' do
            foreach v' adjacent to v in G do
                if parent[v'] == −1 then
                    parent[v']←v;
                    Q'←Q' + {v'};
    synchronization;
    Q ← Q';
```

The static and dynamic scheduling algorithms are both shown in Algorithm 2. The only difference between the two algorithms is how the chunks are assigned to the threads.

OpenMP also has a guided scheduling policy. The guided scheduling policy is similar to the dynamic scheduling policy except that it allows on-the-fly changing of the chunk sizes. The policy starts with large chunk sizes and gradually reduces them. Eventually the policy will reduce the chunk size to 1 and keep it stable. Guided scheduling is not used because it is very similar to dynamic scheduling and the variation of chunk sizes has minor impact on the energy consumption.

3.3 BFS Algorithm in the Fine-Grain Model

As explained in Section 3.1, the BFS algorithm needs the fine-grain execution model to exploit the inter-loop locality. In Algorithm 1, the program repeatedly executes the parallel *for* loop to enlarge the BFS tree. When the loop iterations of a parallel *for* loop are executed, they put new vertices (v' in the algorithm) into Q' for the search in the next parallel *for* loop.

Therefore, the loop iterations in different loops have temporal locality.

When each loop iteration is presented as a codelet in the codelet model, those codelets can pass data via local storage if they are scheduled to the same core. This exploits the locality among codelets in different loops.

However, the scheduling of codelets is not free. The scheduling needs a codelet runtime to maintain the dependency information and assign the codelets to available cores. When the codelet is as small as a loop iteration, the codelet runtime overhead can be quite heavy. Therefore, the following approaches are used to reduce the overhead. Since the dependencies in the parallel *for* loop are quite simple, synchronization is used to guarantee that the dependencies are satisfied. This method eliminates the cost of maintaining the dependency information in the codelet runtime. The codelets can also be assigned without extra cost. If several codelets have temporal locality, they can be scheduled to the same core by using the local storage to buffer the data. If two codelets have no locality, it does not matter if they are assigned to different cores or the same core. Therefore, either static or dynamic scheduling can be used in the codelet execution model.

Therefore, a BFS algorithm was developed using the fine grain-execution model shown in Algorithm 3. This algorithm highlights the exploitation of the inter-loop locality because it is a unique feature of the fine-grain execution model. Local buffers Q_L and Q'_L are used to locally pass data from one loop iteration to another in a different loop.

Algorithm 3 can also be viewed as a hybrid algorithm of the fine-grain and coarse-grain execution models. The algorithm design level uses the codelet model to exploit the inter-loop locality. Then, the implementation level uses synchronization (a typical coarse-grain approach) to reduce the overhead in the codelet runtime.

Algorithm 3: **BFS algorithm pseudo code for the fine-grain execution model.**

input : Undirected graph G and starting vertex $root$
output: A BFS tree represented by a vector $parent$ that stores the parent of each vertex in the BFS tree
Data: Q is the current search list (a global shared queue), Q' is the search list for the next turn (another global shared queue), and each thread has Q_L and Q'_L as local buffers (two local queues)

PSEUDO CODE:

```
foreach element e of parent do e ← −1;
parent[root] ← root;
Q←{root};
(on each thread) Q_L ← ∅;
while (Q ∪ Q_L of each thread) ≠ ∅ do
    Q' ← ∅;
    (on each thread) Q'_L ← ∅;
    foreach v ∈ Q_L in parallel do
        foreach v' adjacent to v in G do
            if parent[v'] == −1 then
                parent[v']←v;
                if Q'_L is full then Q'←Q' + {v'};
                else Q'_L ← Q'_L + {v'};
    foreach v ∈ Q in parallel do
        foreach v' adjacent to v in G do
            if parent[v'] == −1 then
                parent[v']←v;
                if Q'_L is full then Q'←Q' + {v'};
                else Q'_L ← Q'_L + {v'};
    synchronization;
    Q ← Q';
```

4 Test Results

Tests were conducted to study the energy efficiency of the BFS algorithm on a many-core architecture that has both on-chip local storage and on-chip shared memory. The tests analyze the following:

(1) How the exploitation of intra-loop locality in the coarse-grain execution model affects the energy efficiency.

(2) How the exploitation of inter-loop locality in the fine-grain execution model affects the energy efficiency.

(3) Compares the BFS algorithms for the coarse-grain execution model and fine-grain execution model.

4.1 Test Setup

Tests were run on the Intel *fsim* simulation platform as introduced in Section 2.2. Since *fsim* is a functional simulator, it cannot provide accurate execution times. Therefore, we do not report on the algorithm performance. However, *fsim* provides accurate dynamic energy consumption measurements based on counts of the executed instructions. Therefore, the following focuses on the dynamic energy consumption of the BFS algorithms.

The graphs included up to 4K vertices and 64K edges. Larger graphs could not be tested due to memory limitations of the simulator. The graphs were generated

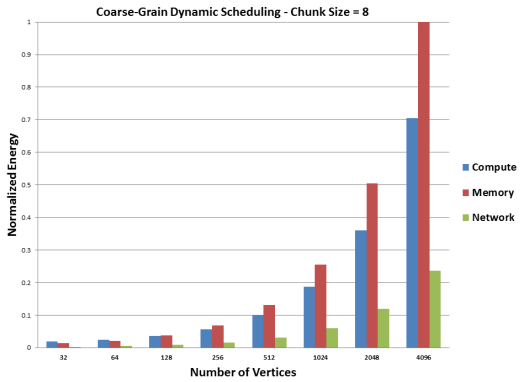

Fig. 4 Energy use by the coarse-grain dynamic scheduling implementation for various graph sizes. The average degree per vertex was 16.

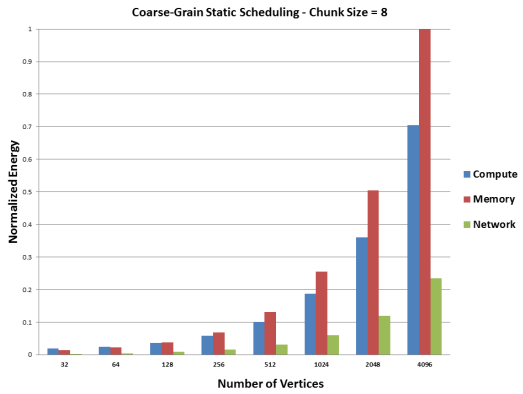

Fig. 5 Energy use by the coarse-grain static scheduling implementation for various graph sizes. The average degree per vertex was 16.

using the RMAT algorithm that follows the parameter setup requirements in the Graph500 specification.

The coarse-grain implementation tested different chunk sizes for both static and dynamic scheduling on a graph with 4096 vertices and an average vertex degree of 16. The energy usage is shown for both scheduling policies with different graph sizes. The fine-grain implementation used three variants with local buffer sizes equal to 32, 64, and 128 elements.

To make the codes extendable to support future larger graphs, each vertex is stored in a 64-bit variable. All the figures are normalized for better reading.

4.2 Major Observations

The test results showed that:

(1) The exploitation of intra-loop locality has minor impact on the energy efficiency (up to 1% reduction of the memory access dynamic energy consumption.)

(2) The exploitation of inter-loop locality reduced the memory access dynamic energy consumption by up to 7%.

(3) The BFS algorithm in the fine-grain execution model is more energy efficient than the BFS algorithm in the coarse-grain execution model.

The results and analyses are explained in detail in the following sections.

4.3 Results and Analysis

4.3.1 Energy Use in the Coarse-grain and Fine-grain Implementations

The first tests evaluated the energy usage of the BFS algorithms in both the coarse-grain and fine-grain

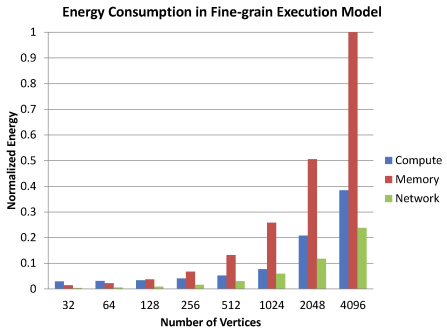

Fig. 6 Energy use by the fine-grain implementation for various graph sizes. The average degree per vertex was 16.

execution models. Fig. 4 and 5 show the test results for the dynamic and static scheduling in the coarse-grain execution model. Fig. 6 shows the results for the fine-grain execution model. The graphs had various sizes from 32 vertices to 4096 vertices. The chunk size in the coarse grain model was 8. The average degree of each vertex in the graph was 16. The energy use shows that:

(1) The implementations are memory-intensive, which is consistent with known results for the BFS problem.

(2) The energy consumption by each part (compute, memory, and network) roughly doubles when the input graph size (number of vertices and number of edges) doubles because the BFS algorithm has linear time complexity as O(M) where M is the total number of edges in the graph. Note that the number of computational instructions and memory access instructions are both proportional to the time complexity.

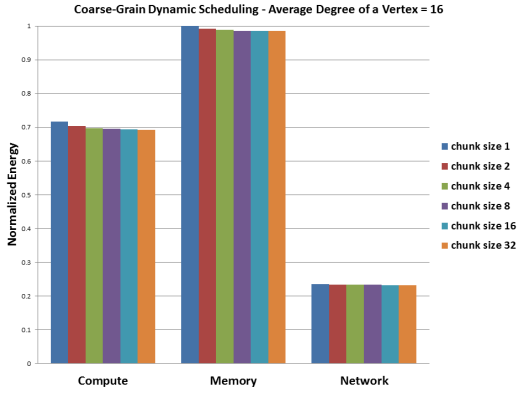

Fig. 7 **Energy consumption by the coarse-grain dynamic scheduling implementation for various chunk sizes. The graph had 4096 vertices with an average degree per vertex of 16.**

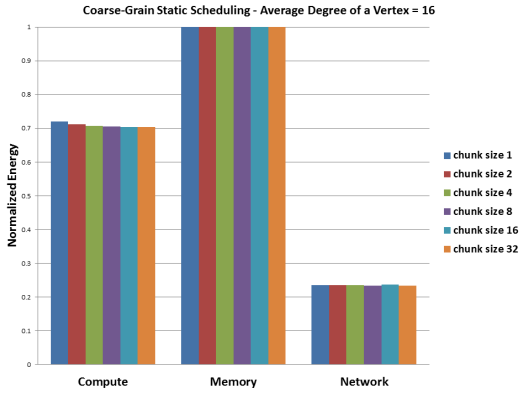

Fig. 8 **Energy consumption by the coarse-grain static scheduling implementation for various chunk sizes. The graph had 4096 vertices with an average degree per vertex of 16.**

Chunk size from 1 to 32 were tested with average degrees of each vertex from 4 to 32. The observations are the same as above.

4.3.2 Comparison between static and dynamic scheduling in the coarse-grain implementation

The energy consumption for both static and dynamic scheduling was measured for the coarse-grain implementation with chunk sizes from 1 to 32. The input graph had 4096 vertices with an average degree per vertex of 16.

Fig. 7 and 8 show the results for the dynamic and static scheduling implementations. The figures show that the chunk size has little impact on the memory access energy use. With the dynamic scheduling, the best result (for a chunk size of 32) reduced the memory access dynamic energy use by 1%. compared to the worst result (for a chunk size of 1). The computational energy saving was slightly better (up to 3% less).

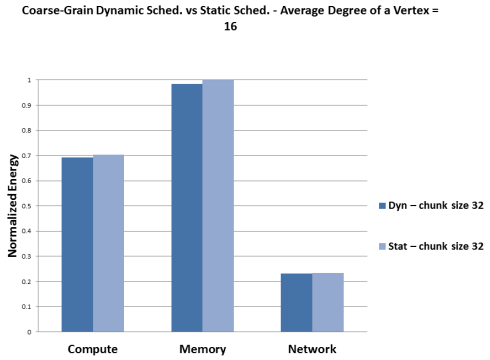

Fig. 9 **Comparison of energy consumption by static and dynamic scheduling for the coarse-grain implementation. The graph had 4096 vertices with an average degree per vertex of 16.**

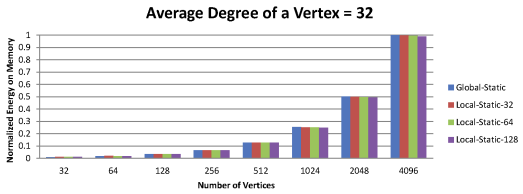

Fig. 10 **Energy consumption for BFS memory access. The average degree per vertex in the input graph was 32. Global-Static is the coarse-grain implementation. Local-Static are the fine-grain implementations with local buffer sizes of 32, 64, and 128. The best input data size for the fine-grain version was 1024 vertices with Local-Static-128 having 2% less energy consumption for memory access than Global-Static.**

Since the BFS is a memory-intensive application, the chunk size has little impact on the dynamic energy consumption. The same results were claimed for sparser graphs. So the data is not reported.

Fig. 9 compares the best cases for the dynamic and static scheduling. The dynamic scheduling is slightly better but the difference is very small (0.9% for the execution energy, 0.4% for the memory energy, and 0.1% for the network energy). Therefore, the major observation is that the scheduling approach has little impact on the dynamic energy consumption.

Therefore, the test results in this section show that the algorithms in the coarse-grain model have little impact on the dynamic energy. It also implies that exploitation of the intra-loop locality has little impact on the dynamic energy.

4.3.3 Comparison of Coarse-grain and Fine-grain Implementations

This section describes how the inter-loop locality affects the energy efficiency by comparing the energy consumption by the BFS implementations using the coarse-grain and fine-grain execution models.

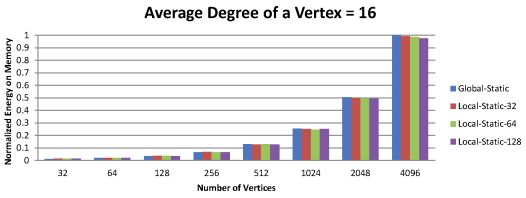

Fig. 11 **Energy consumption for BFS memory access. The average degree per vertex in the input graph was 16. Global-Static is the coarse-grain implementation. Local-Static are the fine-grain implementations with local buffer sizes of 32, 64, and 128. The best input data size for the fine-grain version was 4096 vertices with Local-Static-128 having 3% less energy consumption for memory access than Global-Static.**

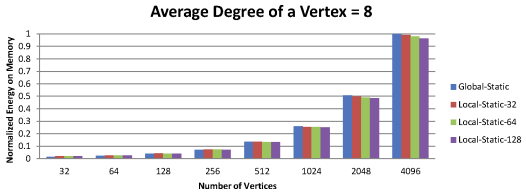

Fig. 12 **Energy consumption for BFS memory access. The average degree per vertex in the input graph was 8. Global-Static is the coarse-grain implementation. Local-Static are the fine-grain implementations with local buffer sizes of 32, 64, and 128. The best input data size for the fine-grain version was 2048 vertices with Local-Static-128 having 4% less energy consumption for memory access than Global-Static.**

Since there is very little difference in the energy consumption rates for the static and dynamic scheduling algorithms on the coarse-grain implementation, the rest of this section uses only static scheduling to represent the coarse-grain version. Since the BFS is a memory-intensive application, the energy use is based on the dynamic energy consumption for memory access for the various cases shown in Fig. 10 to 13. These configurations used input graph sizes from 32 to 4096, graph densities from 32 to 4 average degrees per vertex, and various versions with static scheduling in the coarse-grain execution model and 3 versions of the fine-grain execution model with local buffer sizes

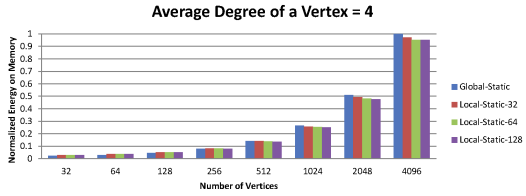

Fig. 13 **Energy consumption for BFS memory access. The average degree per vertex in the input graph was 4. Global-Static is the coarse-grain implementation. Local-Static are the fine-grain implementations with local buffer sizes of 32, 64, and 128. The best input data size for the fine-grain version was 2048 vertices with Local-Static-128 having 7% less energy consumption for memory access than Global-Static.**

from 32 to 128. Since the intra-loop locality has little impact on the energy consumption, the implementations in this section only use inter-loop locality. The static scheduling implementation in the coarse-grain model does not exploit inter-loop locality. In the fine-grain execution models, the exploitation of the inter-loop locality increases as the size of the local buffer increases.

The results show that:

- Greater exploitation of the inter-loop locality reduces more energy use. From all the 4 figures, the energy consumption decreases as the local buffer size increases for graphs with 512 or more vertices.

- Sparser the graphs result in greater energy savings. For example, for the graph with an average degree per vertex of 32, exploitation of the inter-loop locality reduces the memory access dynamic energy use by 2%. However, for the graph with an average degree per vertex of 4, exploitation of the inter-loop locality reduced the dynamic energy use by 7%. This is because BFS trees in sparser graphs are normally higher, which gives more opportunities to exploit the inter-loop locality.

- All the fine-grain implementations use less energy than the coarse-grain implementation.

5 Related Work

The BFS algorithm has been studied for many years because it is a fundamental graph algorithm that is widely used in many applications such as social network analyses [11] and path planning [12].

There have been many studies of distributed BFS algorithms [13–15]. In recent years, Bader and Madduri [16] studied the BFS implementation on a large scaled graph that achieves significant speedup on MTA-2. Scarpazza *et al.* [8] studied how to effectively employ the Cell Broadband Engine to perform BFS on large graphs. John *et al.* [17] described an efficient BFS algorithm for abstract architectures that used a tree-structured memory model. BFS has also been implemented on the Intel Nehalem architecture [18, 19] and large scale distributed memory systems [20, 21].

Those studies of the BFS algorithms have focused on performance and scalability. The major difference between the present work and those previous studies is

that we focus on the exploitation of data locality and how it affects the energy efficiency.

Currently, there are only a few studies of the energy efficiency for the BFS problem. Satish *et al.* [6] claimed their work to be "the first paper showing energy efficiency on the Graph500 benchmark". They applied their BFS optimization on a Intel architecture with 3-level caches. However, the present BFS algorithm targets architectures with local storage but no cache.

6 Conclusion

This paper shows how to exploit data locality in the BFS application. The paper shows that the traditional OpenMP-like execution models are unable to exploit the inter-loop data locality that reuses data between loop iterations of different loops. The inter-loop locality can be exploited in a fine-grain execution model such as the Codelet Model. A BFS algorithm is then described using the Codelet Model to exploit the inter-loop locality.

Tests are run on a simulation platform developed by Intel for the UHPC project [10] for the design of future extreme-scale architectures. The results show that this BFS algorithm reduces dynamic energy use by up to 7% for memory accesses compared to a BFS implementation based on OpenMP loop scheduling.

Acknowledgements

This research was made possible by the generous support of the NSF through grants CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534. This research was also based upon work supported by the Department of Energy (National Nuclear Security Administration) under the Award Number DE-SC0008717. Moreover, this work was partly supported by European FP7 project TERAFLUX, id. 249013.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, ExaScale Computing Study: Technology Challenges in Achieving ExaScale Systems, DARPA IPTO, Air Force Research Labs, Tech. Rep., Sep. 2008.

[2] J. Torrellas, Architectures for extreme-scale computing, *Computer*, vol. 42, no. 11, pp. 28–35, Nov. 2009.

[3] IBM Cell Broadband Engine, `http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`.

[4] J. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, Tiny threads: A thread virtual machine for the cyclops64 cellular architecture, presented at the Fifth Workshop on Massively Parallel Processing, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPS 2005), Denver, CO, USA, 2005.

[5] R. Knauerhase, R. Cledat, and J. Teller, For extreme parallelism, your os is sooooo last-millennium, In *Proc. of the 4th USENIX conference on Hot Topics in Parallelism*, ser. HotPar'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3.

[6] N. Satish, C. Kim, J. Chhugani, and P. Dubey, Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing, In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 14:1–14:11.

[7] The graph 500, list.http://www.graph500.org/, 2010–2012.

[8] D. P. Scarpazza, O. Villa, and F. Petrini, Efficient breadth-first search on the cell/be processor, *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1381–1395, Oct. 2008.

[9] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, Using a "codelet" program execution model for exascale machines: position paper, In *Proc. of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 64–69.

[10] DARPA-BAA-10-37, UHPC: Ubiquitous high performance computing, `http://www.darpa.mil/Our_Work/MTO/Programs/Ubiquitous_High_Performance_Computing_(UHPC).aspx`, Arlington VA, USA, 2010-2012.

[11] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, Measurement and analysis of online social networks, In *Proc. of the 7th ACM SIGCOMM conference on Internet measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 29–42.

[12] A. Sud, E. Andersen, S. Curtis, M. Lin, and D. Manocha, Real-time path planning for virtual agents in dynamic environments, In *ACM SIGGRAPH 2008 classes*, ser. SIGGRAPH '08. New York, NY, USA: ACM, 2008, pp. 55:1–55:9.

[13] B. Awerbuch and R. Gallager, A new distributed algorithm to find breadth first search trees, *Information Theory, IEEE Transactions on*, vol. 33, no. 3, pp. 315 – 322, may 1987.

[14] G. Y. Ananth, V. Kumar, and P. Pardalos, Parallel processing of discrete optimization problems, in *IN ENCYCLOPEDIA OF MICROCOMPUTERS*. Marcel Dekker Inc, 1993, pp. 129–147.

[15] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, A scalable distributed parallel breadth-first search algorithm on bluegene/l, In *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 25.

[16] D. A. Bader and K. Madduri, Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2, In *Proc. of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530.

[17] T. St. John, J. B. Dennis, and G. R. Gao, Massively parallel breadth first search using a tree-structured memory model, In *Proc. of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 115–123.

[18] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, Scalable graph exploration on multicore processors, In *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[19] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency, in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, may 2012, pp. 378 –389, doi: 10.1109/IPDPS.2012.43.

[20] A. Buluç and K. Madduri, Parallel breadth-first search on distributed memory systems, In *Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 65:1–65:12.

[21] K. Ueno and T. Suzumura, Highly scalable graph search for the graph500 benchmark, In *Proc. of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012, pp. 149–160.

Chen Chen received his PhD in Computer Science and Technology from Tsinghua University in 2011. Currently he is a post-doctoral researcher at University of Delaware. His thesis was about memory consistency models for many-core architectures. His research interests lie at computer architecture, memory consistency model, and parallel execution model.

Souad Koliai received her PhD in Computer Engineering from the University of Versailles (France) in July 2011. Currently she is a post-doctoral researcher at CAPSL at University of Delaware. Her thesis was about developing systematic performance analysis approaches for scientific programs. Her research interests lie at computer architecture, parallel programing, performance analysis, and code optimization.

Guang R. Gao received the MS and PhD degrees in electrical engineering and computer science from Massachusetts Institute of Technology, in 1982 and 1988, respectively. He is currently an endowed distinguished professor of electrical and computer engineering at the University of Delaware, Newark. He is a fellow of the IEEE and the ACM. Prior to joining the University of Delaware, he served on the faculty of the School of Computer Science, McGill University at Montreal. He has founded and directed the Computer Architecture and Parallel Systems Lab (CAPSL), University of Delaware. His main research interests include high-performance computer systems: architectures, programming models, compilers, and system software, and their applications. He has published more than 200 papers in international journals and conferences. He has co-initiated several important conferences (such as PACT and CASES) and has served as a program committee member in many high-quality international conferences and workshops. He has served as an editor or as a member of the editorial boards of the IEEE Transaction on Computers, IEEE Concurrency, and IFIP Parallel Processing Letters.