Just-In-Time Locality and Percolation for Optimizing Irregular Applications on a Manycore Architecture*

Guangming Tan[1,2],Vugranam C. Sreedhar[3], Guang R. Gao[2]

[1] Department of Electrical and Computer Engineering, University of Delaware
[2] Key Laboratory of Computer System and Architecture, Institute of Computing Technology,Chinese Academy of Science
[3] IBM T. J. Watson Research Center. USA
{guangmin, gao}@capsl.udel.edu, vugranam@us.ibm.com

Abstract. This paper presents a new technique to optimize locality of irregular programs by leveraging parallelism on a massive many-core architecture – IBM Cyclops64 (C64). The key idea is to achieve *Just-In-Time Locality* which ensures that data are available *locally* for computation to use. The proposed percolation model for Just-In-Time Locality moves data proactively close to the computation and organizes the data layout such that locality is exploited effectively. The percolation model opens a door for exploiting locality through parallelism, which is an advantage of the future many-core architecture. We implemented the percolation strategy in the context of two irregular applications on C64. Our experimental results are very encouraging and we get an order of magnitude improvement in performance of irregular applications. We also drastically improve the scalability of the applications that we studied.

1 Introduction

Emerging future microprocessor chip technology unveils a new generation of many-core chip architecture that may contain 100 to 1,000 processing cores. In order to improve the performance and scalability of large-scale applications computer architects, system software designers and application scientists are realizing that they must work closely together to investigate how to exploit the computational power of such new many-core architecture. At a high level there are two kinds of applications—"regular applications" where data access and control flow follow regular and (statically) predictable patterns, and "irregular applications" where data access and control flow have statically (and often even dynamically) unpredictable patterns. Many irregular applications are often implemented using complex pointer data structures such as graph and queue, and recursive control flow is often used to traverse and manipulate such complex pointer data structures. It is difficult and often impossible to capture the data access patterns at compilation time for such applications. For architectures that support memory

* This work has been performed when the first author is a visiting scholar at Computer Architecture and Parallel System Laboratory (CAPSL) of University of Delaware. He is currently associated with Institute of Computing Technology.

hierarchy, unpredicatable data access patterns often lead to higher off-chip memory access latency, which in turn can degrade the performance and scalability of such irregular applications.

The current threading library (e.g., Pthreads), a combination of compiler directives and libraries (e.g., OpenMP) and optimistic parallelization [1–3] were not designed to support programming for tolerating off-chip latency, or to handle efficient allocation and movement of data across hierarchy levels. It is often the case that in the underline thread execution model, a thread is enabled and activated as soon as all data and control dependencies are satisfied. Such thread execution models may do well for regular applications, where there is an inherent memory locality in the application. Unfortunately, irregular applications often do not have inherent memory locality and so the weaker model often performs poorly.

In this paper we exploit several characteristics of IBM Cyclops64 (C64) architecture [4] and its runtime threading model to drastically improve the scalability and performance of irregular applications. Our runtime threading model consists of two phases: (1) memory access phase and (2) computation phase. These two phases are orchestrated using Just-In-Time Locality and percolation model in such a way that we can amortize the latency of accessing non-local data across multiple hardware threads. The main contributions of this paper are as follows:

- We highlight the basic notion of Just-In-Time Locality - that has been studied conceptually in our past work on percolation model under the HTMT project [12, 14] to improve and exploit data locality in irregular applications. We show how to interleave computation and memory access such that a thread is enabled only when all of its data, control, and *locality* constraints are satisfied. To hide the latency of memory access we overlap and pipeline the computation phase and the memory access phase across multiple cores or hardward threads.
- We describe percolation programming technique to optimize two important irregular applications—the betweenness centrality algorithm and the dynamic programming algorithm.
- We have implemented our approach on C64 and using our approach we obtained a performance improvement of 4-50 times for betweenness centrality and of 1-2 times for dynamic programming.

The rest of the paper is organized as follows: In section 2, we propose the percolation programming technique in detail. Section 3 discusses how to program irregular program with percolation. Section 4 evaluates the performance of applying percolation model to two irregular applications – betweenness centrality and dynamic programming on a many-core chip architecture. In section 5, we discuss the existing related techniques. Finally, section 6 concludes this paper.

2 Percolation Model and Just-In-Time Locality

In this section we describe some of the key ideas behind our percolation model by exploiting some of the key characteristics of C64 architecture [4].

2.1 C64 Architecture

C64 is a many-core chip architecture that employs a large number of hardware thread units (processing cores), half as many floating point units, (on-chip) SRAM memory banks, an interface to the off-chip DDR SDRAM memory and bidirectional inter-chip routing ports. The C64 chip has no data cache and features a three-level memory hierarchy (Scratchpad memory, on-chip SRAM, off-chip DRAM). A portion of each thread unit's corresponding on-chip SRAM bank is configured as the scratchpad memory. Therefore, the thread unit can access its own scratchpad memory with very low latency through a "backdoor", which provides a fast temporary storage to exploit locality under software control. The remaining portion of each on-chip SRAM bank, together, forms the on-chip global memory that is uniformly addressable from all thread units. There are 4 off-chip memory controllers connected to 4 off-chip DRAM banks.

C64 incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through the execution of a special instruction (casing a direct hardware interrupt). All the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers. C64 provides no resource virtualization mechanism: the thread execution is *non-preemptive* and there is no hardware virtual memory manager. The former means the OS will not interrupt the user thread running on a thread unit unless the user explicitly specifies a termination or an exception inside C64 chip that is *visible* to the programmer. From a programming model perspective such non-preemptiveness implies that it is important not to stall an execution of a thread once it is scheduled on to a hardware thread unit. In the rest of this section we will describe our percolation model that avoids such unnecessary stalls of running threads.

2.2 Percolation for Just-In-Time Locality

In our percolation threading model a thread has to satisfy two requirements before it can be enabled and ready to run: (i) data/control dependencies have to be satisfied and (ii) *locality* constraints have to be satisfied. The latter requirement essentially enforces that all data referenced by a thread should be local before a thread can be scheduled to run on a hardware thread unit. We represent a program as a directed acyclic task graph, where each node is a task, and a direct arc between two nodes represents a precedence relation between tasks. For regular programs, a user (or a compiler) can often automatically identify computation and memory access tasks, and schedule them such that the latency of memory accesses incurred by the memory access/movement tasks are tolerated. But for an irregular program like graph traversal, it is often difficult to automatically identify task level parallelism and their data access patterns to perform such latency tolerant scheduling statically. One solution is to let the users explicitly specify the task level parallelism. Recall that a necessary condition for a task to become enabled is that all data required by a (computation) task have been produced, and all control dependences are satisfied. In a task graph, a node s (i.e., a task) is enabled if all its predecessor nodes have completed and the required data and control dependences have be satisfied. We call a task that satisfies data and control dependence requirements

as being *logically enabled*. In our percolation model it is not sufficient for a logically enabled task to run. A logically enabled task often cannot immediately run since the data may still be in off-chip memory hierarchy or in the local memory of other cores. We introduce locality constraints in addition to data and control dependence requirements to overcome the latency gap through memory hierarchy. We call a logically enabled task as *locality enabled* if it also satisfies locality constraints. For a task to be locality enabled all data referenced by the task should become local before a task can begin execution. The locality requirement ensures that the corresponding code and data of the candidate task are resident in the same level of memory hierarchy where it is to be enabled.

C64 also supports explicit memory hierarchy and so we use multigrain parallelism to improve performance and scalability of applications. Coarse-grained parallelism is used to enable a thread at coarse-grain level where data and control dependencies are satisfied, and fine-grain parallelism is used to enable a thread at fine-grain level when locality constraint is satisfied. The coarse-grain tasks reflect logical parallelism in the user program. It is easy to map each task to an independent thread (core) if the depended data is available in a shared memory space. Our percolation model creates additional fine-grain parallelism within each coarse-grain task. It is important to note that our percolation-based multi-grain parallelism is different from conventional multi-grain parallelism techniques that are used to combine task and data parallelism [5]. The fine-grain parallel tasks in the percolation model are exploited as separating memory access from computation operations within a coarse-grain task. The advantages of separating memory from computation are: (1) in addition to the fine-grain parallelism, we can also pipeline the different phases of memory access and computation tasks to hide the overhead of memory access, and (2) it provides an opportunity to elaborate the memory access tasks so that they are aware of memory hierarchy and transform non-linear memory access with high latency to linear memory access with low latency. A separate memory access task may reorganize (gather) the dispersed references in advance in the pipeline of tasks. Within memory hierarchy, the tasks for locality requirement may involve either collecting the data toward the cores where the task is enabled, called inward percolation, or sending/migrating the data away from the cores, called outward percolation. A percolation task scheduler causes the data to meet the corresponding threads just in time at the vicinity of the cores where the computation task is to be carried out.

2.3 Discussion

The key idea behind percolation model is to bring data close to computation just in time so that the computation thread can run to completion. For non-preemptive thread units such as in C64, it is important to reduce the number of stalls during execution. In percolation model rather than delaying or suspending a thread during execution, we avoid scheduling such threads that do not have locality constraints satisfied. Percolation model is closely related to prefetching, except that in prefetching, prefetch instructions are inserted within a thread that is ready to be scheduled, and the hope is that at the time the thread needs the data it will be available for use. Often prefetching instructions are either inserted too soon (in which case the prefetched data is evicted from the cache) or too late. In both cases the corresponding thread will stall the (non-preemptive) thread

unit on which it is running. Since C64 has no caches nor does it support preemptive hardware thread units, delaying or suspending active threads or prefetching is not well suited to improve performance and scalability of applications.

3 Percolation Programming

In this section, we discuss in detail two irregular applications (betweenness centrality with graph traversal in SSCA2 [6] and non-serial polyadic dynamic programming (DP) [7]) and show how to program them for percolation. There are three parts to percolation programming: (1) Collect data from non-contiguous locations just in time to obtain Just-In-Time Locality in the on-chip memory for the computation phase; (2) Compute the relevant information based on Just-In-Time Locality on-chip data; and (3) Finally, map the information thus computed back to off-chip memory. The first and last phase form the memory tasks and are mapped to helper threads by the runtime system. The second phase is the main computation phase that is mapped to a computation thread. Due to space limitation, we only present the pesudo code of SSCA2. It is important to keep in mind that the memory tasks and computation tasks are pipelined by the runtime system to reduce the critical path. Also, it is upto the programmer to create coarse grain parallelism (that includes multiple computation tasks and memory tasks) depending on what is being computed within the application.

3.1 Percolation Programming for SSCA2

In this section we describe our percolation programming for SSCA2. Recall that there are two phases in SSCA2: the BFS phase (See Figure 1) and BT phase [6]. To simplify the presentation we only describe the percolation programming for the BFS phase.The first step in percolation programming is to identify the memory tasks. Let us denote the set of the vertices that is being extended in the current queue (the ith level of BFS tree) as $V_i = \{v_{i1}, v_{i2}, ..., v_{ik}\}$. Let $N_j = \{w_{j1}, w_{j2}, ..., w_{jk_j}\}, 1 \leq j \leq k$ denote the neighboring set of vertices of a vertex v_{ij}. According to the algorithm, the unvisited neighbor vertices w ($d[w] = -1$) is added to the current queue and the vertices that is being extended in the shortest path ($d[w] = d[v] + 1$) is added to the set of predecessor $P[w]$. Note that the layout of these N_j in the adjacency array may be non-contiguous and have variable stride. Note that if we compact all the neighbors in one level into one large set: $UN_i = \bigcup_{1 \leq j \leq n} N_j$, the compacted non-contiguous memory region can be considered as a contiguous linear array so that it is easy to partition it among parallel threads. In our implementation, we do *not* explicitly perform such a compaction operation in the off-chip memory, but inside we use the Just-In-Time Locality principle.

The inward percolation consists of computing the start address and size of the neighboring vertices region in adjacency array of each vertex, and collecting neighboring vertices that is dispersed in the off-chip memory address (adjacency array) into a contiguous on-chip memory address. We also collect the corresponding elements in d, σ into a contiguous on-chip memory address. Notice that there is a producer-consumer relationship between the collection of neighboring vertcies and collection of d, σ. Also, the memory references of d, σ are discrete because the distribution of the neighboring

```
1  BFS(int v) {
2    int dv = d[v]; //length of the shortest path
3    int sigmav = sigma[v]; //the number of the shortest path
4    for (i = 0; i < NumEdges[v]; i++) {
5      w = Adjacent[index[v]+i];
6      if (d[w] < 0) {
7        d[w] = dv + 1;
8        sigma[w] = 0;
9      }
10     if (d[w] = dv + 1)
11       sigma[w] = sigmav + 1;
12   }
13 }
```

Fig. 1. The sequential BFS codes without percolation

vertices obeys a law of power in a scale free graph. Due to this property of scale-free graph we believe that the thread speculation techniques are not effective in practice. Once we compute the relevant information (d, σ) we write them back to off-chip memory using yet another memory task.

The complete parallel percolation program for BFS phase is shown in Figure 2. Once the programmer defines the coarse-grain parallel tasks the runtime system automatically divides these tasks into multiple sub-tasks and pipelines them. Obviously, the dependence between the sub-tasks is inherited from that specified in the user program. In order to achieve the pipeline of computation, inward and outward memory tasks, the union set UN_i is partitioned into multiple sub-blocks. When computation tasks are processing the data in block i, inward percolation memory tasks gather the data in block $i+1$ and the outward percolation memory tasks scatter the results that are generated using the data in block $i-1$.

3.2 Percolation Programming for DP

The dynamic programming (DP) algorithm in RNA secondary structure prediction [8] belongs to a type of non-serial polyadic dynamic programming [7]. We can use a simple recursive formulation to represent the computation:

$$m[i,j] = \begin{cases} min_{i \le k < j}\{m[i,j], m[i,k] + m[k+1,j]\} & 0 \le i < j < n \\ a(i) & i = j \end{cases} \tag{1}$$

The irregular behavior comes from the non-consecutive data dependence and irregular iteration domain which is a triangular space. Basically, we could use a blocking strategy to fill the matrix. In [9], the computation of a block is decomposed into combination of the depended blocks. According to the dependence, the computational sequence of blocks is from down to up and from left to right in the matrix, and each block depends on the blocks on both the same row and the same column. When both $A(0,1)$ and $A(1,3)$ are combined to calculate $A(0,3)$, an ideal memory layout should look like: $A(0,1)$ is row-wise and $A(1,3)$ is column-wise. One strategy of block data layout is to store each block as both row wise and as column wise array layout. However, this

```
/*three pipelined phase: (1) off-chip memory read;
(2) compuation (accessing on-chip memory);
 (3) off-chip memory write.*/
BFS(int v) {
  int offset = 0;
  int turn = 0;
  int dv = d[v];
  int sigmav = sigma[v];
  SPAWN_TASK{
  for (i = 0; i < buffsize; i++)
        buff[turn][i] = Adjacent[index[v]+offset+i];
   offset += buffsize;
   turn ^= 1;};
  BARRIER_WAIT();
  while (offset < NumEdges[v]) {
    //1. off-chip memory read
    SPAWN_TASK{
    for (i = 0; i < buffsize; i++)
        buff[turn][i] = Adjacent[index[v]+offset+i];
    offset += buffsize;
    turn ^= 1;};
    SPAWN_TASK{
    for (i = 0; i < buffsize; i++) {
         w = buff1[i];
         buff2[turn][i] = d[w];
         buff3[turn][i] = sigma[w];
     }
     turn ^= 1;};
     //(2). compuation (accessing on-chip memory);
     SPAWN_TASK{
     for (i = 0; i < buffsize; i++) {
        if (buff2[turn][i] < 0) {
                buff2[turn][i] = dv+1;
                buff3[turn][i] += 0;
        }
        if (buff2[turn][i] == dv+1)
                buff3[turn][i] += sigmav;
     }
     turn ^= 1;};
     //(3). off-chip memory write
     SPAWN_TASK{
     for (i = 0; i < buffsize; i++) {
        w = buff[turn][i];
        d[w] = buff2[i];
        sigma[w] = buff3[i];
      }
      turn ^= 1;};
      BARRIER_WAIT();
   }
}
```

Fig. 2. BFS codes with percolation on IBM C64

doubles the memory usage which is not practical. We assume that the matrix is stored as a row-wise linear array. Thus, the stride of accesses in different row or column within each block is not constant.

According to the data dependence, the computation of block $A(i,j)$ needs to access other blocks $A(i,i), A(i,i+1), ..., A(i,j-1)$ and $A(i+1,j), A(i+2,j), ..., A(j,j)$. For example the program accesses $< A(0,0), A(0,3) >$, $< A(0,1), A(1,3 >$, $< A(0,2), A(2,3) >$ and $< A(0,3), A(3,3) >$ during the calculation of $A(0,3)$. We assume that the triangular DP matrix is stored as a linear array in off-chip memory. The percolation transformation is responsible to transform the non-contiguous access of a block into a contiguous access in on-chip memory. When a block $A(i,j)$ is computing, the percolation threads gather the non-contiguous elements in off-chip memory into multiple contiguous space in on-chip memory, then scatter the results to the corresponding locations in off-chip memory. The pipeline achieves Just-In-Time Locality, and the percolated data will be used immediately by the computation task. The unused data will never exist in on-chip memory at that time even if the spatial locality of cache mechanism is satisfied. For the example of computing block $A(0,2)$, when the program is percolating block $A(1,2)$ into on-chip memory, a conventional spatial locality optimization strategy or speculation may load the elements in block $A(1,3)$. Obviously, the current computation does not need $A(1,3)$ at all. Since C64 provides user programmable scratchpad, the runtime system ensures that such unnecessary data is not loaded into the on-chip memory.

4 Evaluation

We have implemented Just-In-Time Locality and percolation for the two programs in our many-core architecture C64 execution-driven simulation platform. The toolchain on C64 consists of an optimized GCC compiler, a thread execution runtime systems TNT [10] (Pthread-like) and a TNT-based OpenMP [11]. In this section we present our empirical results and compare them with the corresponding OpenMP programs on C64.

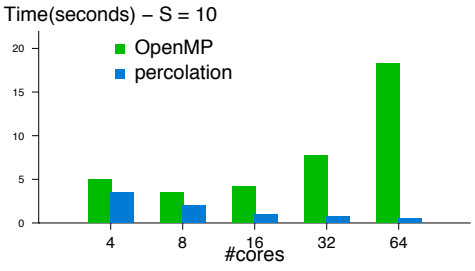


Fig. 3. Performance comparison of percolation with OpenMP. Left bars are openmp and the right bars are percolation.

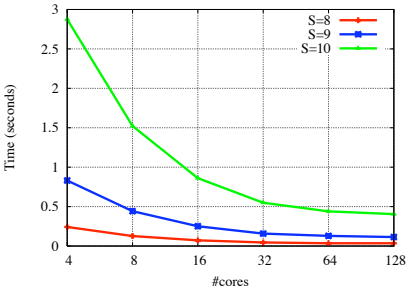


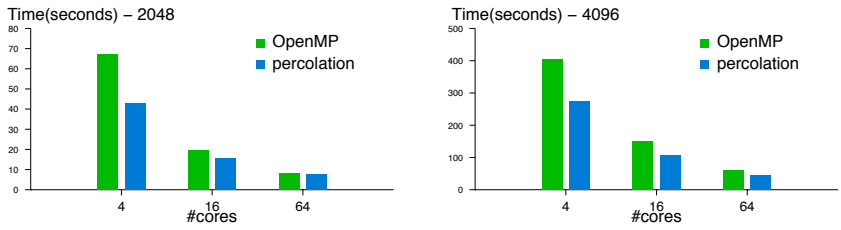
Fig. 4. Scalability of parallel betweenness centrality algorithm. The number of edges $E(n) = 8n$.

4.1 Empirical Results for SSCA2 with Percolation

For SSCA2 we represent the problem size in term of S, where the number of vertices is 2^S. Comparing the result with the OpenMP implementation, we can see that the percolation process shows a significant performance and scalability improvements (see Figure 3). Figure 4 illustrates the performance and scalability as we increase the number of threads for three different scales (i.e., the problem size). Using our approach we achieve almost linear speedups for all test cases when the number of threads is less than 32. For the test case with a problem size $S = 8$, the performance stops increasing when the number of threads reaches 128 because the number of available parallel sub-tasks is less than the number of hardware thread units. However, we improve the performance when the problem size is increased, i.e for $S = 9$ and 10. The degree of a vertex determines the amount of parallelism that we can exploit. In percolation programming we leverage multi-grain parallelism to reduce the number of idle threads. On the other hand the maximum degree of a vertex is 64 for problem size $S = 8$. So the available parallelism for this small problem size leads to a smaller performance on 128 threads. For $S = 9$ and 10, where the maximum vertex degrees are 94 and 348, we further improve the performance and scalability of the application.

4.2 Empirical Results for DP with Percolation

In order to highlight the advantage of separation of computation from memory operations, we compared with the performance of a baseline program implemented with directly blocking algorithm by OpenMP. As shown in Figure 5(a) and 5(b), the reconstructed program with percolation reduces the execution time for matrix sizes of 2048×2048 and 4096×4096. The effect of percolation for DP is not so significant with that for SSCA2. Note that in the implementation of DP we use blocking technique to re-organize the DP matrix so that it shows more inherent locality, which can not be observed in the irregular execution of SSCA2. Further, our percolation program improves the scalability slightly. Figure 6 reports the absolute speedup achieved by percolation and OpenMP programs.



(a) matrix size = 2048×2048 (b) matrix size = 4096×4096

Fig. 5. Performance comparison of percolation with OpenMP. Left bars are OpenMP and the right bars are percolation.

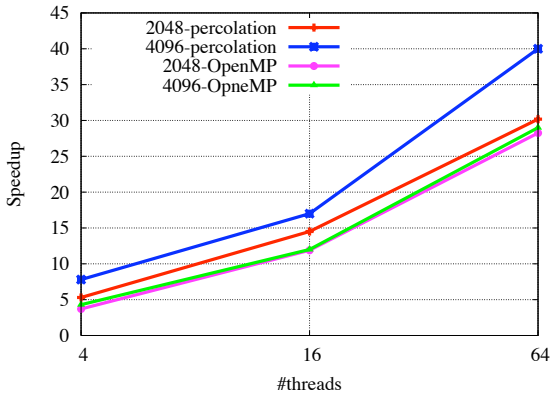


Fig. 6. Scalability results for matrix sizes of 2048×2048 and 4096×4096.

5 Related Work

The percolation model has its deep root in the HTMT execution model proposed well over a decade ago as the basis of the world first (to the best of our knowledge) petaflops architect project [12]. The concept of percolation was developed early under HTMT project, and was first exposed in [13, 14]. Unlike the HTMT time, The work reported here is partly motivated by the challenging technology trend of modern large-scale many-core chip technology, and driven by the productive software technology available to us that is not available during HTMT project due to resource limitations.

In our parallel pipelining algorithm we overlap computation task with memory task. The concept of overlapping computation with I/O, network, and other long latency operations is an old concept. Prefetching techniques [15–17] and thread speculation [1, 18, 19] also use such overlapping concept. Most previous work on prefetching also focused on moving data (mostly contiguous data) from main memory to local memory (either to register or cache) prior to execution. In the previous prefetching or speculation, conceptually computation threads ”pull” the data locally using prefetch instructions. In our method the local data determines which computation thread is ready to execute. In other words, data that is local to a core will ”pull” computation thread to execute on the core. In prefetching there is no control on how much data to prefetch—prefetching too much or too less data can impact the performance. Besides, previous works do not discuss the impact of prefetching in the context of massive multithreading many-core. A variant of thread level speculation uses dependences by monitoring the reads and writes to memory locations. In producer-cosumer loop iterations, the speculative execution leads to a violation of dependence, then must roll back.

There have been several work on the optimization of irregular programs on parallel architectures. Recently Erez et.al [20] performed a comprehensive study of 4 irregular

scientific computing applications on a streaming processor. Both their work and ours share the streaming programming style of gather-compute-scatter. The way to gather data ahead makes our approach different from theirs. In [20] the streaming processor uses a DMA-style transfer, our approach utilizes the ample hardware thread units, where to hide the overhead of transformation is easier and requires less hardware cost.

6 Conclusion

Both computer architects and system developers is yet to evaluate the new many-core architectural features and show how such features can be effectively exploited when executing challenging irregular applications like graph traversal and dynamic programming in practice. This paper introduces Just-In-Time Locality and percolation model for improving the locality of irregular applications on C64 many-core architecture. However, our percolation model is not uncontroversial. For example, we did not discuss the role of ”reusability” of data to be percolated. It is obvious that we should try to give higher priority to data that can have better reuse. Another is connection to load balancing: one obviousely need to coordinate the percolation (and Just-In-Time Locality) with the place where a thread is likely to be scheduled for execution. In an irregular code, we cannot expect load is evenly distributed among the processing cores - and runtime coordination of data movement and load balancing should be smooth. We can imagine cases where not all cores in a 100+ core chip can be kept usually busy all the time - as our experience has been for C64. In this case, some of the idle cores should be employeed to assist the percolation and coordination - an interesting research topic by its own right. Finally one fair doubt is the impact on percolation model on programmability - a question that does not has a short answer and we will have to leave it for future work.

7 Acknowledgment

We would like to thank all reviewers and the shepherd for improving this paper. The authors would like to acknowledge Russo Andrew and Ge Gan at CAPSL for their help. This work is partially supported through the support from NSFC (60633040), IBM, ET International, the National Science Foundation (CNS-0509332).

References

1. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: Proceedings of the 27th Annual International Symposium on Computer Architecture. (2000)
2. Rauchwerger, L., Zhan, Y., Torrellas, J.: Hardware for speculative run-time parallelization in distributed shared memory multiprocessors. In: Proceedings of the 4th International Symposium on High-Performance Computer Architecture. (1998) 162
3. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, P.: Optimistic parallelism requires abstractions. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. (2007) 211–222

4. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In: The 34th International Symposium on Computer Architecture. (2007)
5. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA (2006)
6. Bader, D.A.: HPCS scalable synthetic compact applications 2 graph analysis. www.highproductivity.org/SSCABmks.htm (2006)
7. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to parallel computing. Addison Wesley (2003)
8. Zuker, M., Mathews, D.H., Turner, D.H.: Algorithms and thermodynamics for RNA secondary structure prediction: A practical guide. Kluwer Academic Publishers (1999)
9. Tan, G., Feng, S., Sun, N.: Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM (2006) 78
10. Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Tiny threads: A thread virtual machine for the Cyclops-64 cellular architecture. In: Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th rnational Parallel and Distributed Processing System. (2005)
11. Cuvillo, J., Zhu, W., Gao, G.R.: Landing OpenMP on Cyclops-64: An efficient mapping of openmp to a many-core system-on-a-chip. In: The 3rd ACM International Conference on Computing Frontiers, Ischia, Italy (2005)
12. Gao, G.R., Likharev, K.K., Messina, P.C., Sterling, T.L.: Hybrid technology multi-threaded architecture,. In: Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation. (1996) 98–105
13. Amaral, J.N., Gao, G.R., Merkey, P., Sterling, T., Ruiz, Z., Ryan, S.: Performance prediction for the HTMT: A programming example. In: TFP3'99. (1999)
14. Gao, G., Amaral, J.N., Marquez, A., Theobald, K.: A refinement of the HTMT program execution model. Technical report, CAPSL,University of Delaware (1998)
15. Wu, Y.: Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM (2002) 210–221
16. Zhang, Z., Torrellas, J.: Speeding up irregular applicaitons in shared-memory multiprocessors: Memory binding and group prefetching. In: 22nd International Symposium on Computer Architecture. (1995)
17. Mowry, T., Gupta, A.: Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. Journal of Parallel and Distributed Computing **12** (1991) 87–106
18. Collins, J.D., Tullsen, D.M., Wang, H., Shen, J.P.: Dynamic speculative precomputation. In: the 34th Annual International Symposium on Microarchitecture. (2001)
19. Zhang, W., Tullsen, D.M.: Accelerating and adapting precomputation threads for efficient prefetching. In: 3th International Symposium on High Performance Computer Architecture. (2007)
20. Erez, M., Ahn, J.H., Gummaraju, J., Rosenblum, M., Dally, W.J.: Executing irregular scientific applications on stream architectures. In: ICS '07: Proceedings of the 21st annual international conference on Supercomputing, New York, NY, USA, ACM (2007) 93–104