Toward a Parallel Turing Machine Model

Peng Qu[1(✉)], Jin Yan[2], and Guang R. Gao[2]

[1] Tsinghua University, Haidian, Beijing, China
shen_yhx@163.com

[2] University of Delaware, Newark, DE 19716, USA

Abstract. In the field of parallel computing, the late leader Ken Kennedy, has raised a concern in early 1990s: "Is Parallel Computing Dead?" Now, we have witnessed the tremendous momentum of the "second spring" of parallel computing in recent years. But, what lesson should we learn from the history of parallel computing when we are walking out from the bottom state of the field?

To this end, this paper examines the disappointing state of the work in parallel Turing machine models in the past 50 years of parallel computing research. Lacking a solid yet intuitive parallel Turing machine model will continue to be a serious challenge. Our paper presents an attempt to address this challenge — by presenting a proposal of a parallel Turing machine model — the PTM model. We also discuss why we start our work in this paper from a parallel Turing machine model instead of other choices.

Keywords: Parallel Turing machine · Codelet · Abstract architecture · Parallel computing

1 Introduction and Motivation

Parallel computing is a type of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into multiple smaller ones such that teamwork by multiple processors can deliver the computational result sooner.

Historically, we have witnessed significant progress in parallel computing from 1960 s to 1980s. Notable advances in parallel computer architecture have been made ranging from the R&D on massive SIMD parallel computers [3] to the vector processing computers. However, in the early 1990s, the once bright looking future of the parallel and high-performance computing field had encountered serious challenge — so serious that had prompted a well known speech entitled

Peng Qu is pursuing his Ph.D. thesis study supervised by Prof. Youhui Zhang at Tsinghua University.

G.R. Gao et al. (Eds.): NPC 2016, LNCS 9966, pp. 191–204, 2016.
DOI: 10.1007/978-3-319-47099-3_16

"Is Parallel Computing Dead?" [30] given by late Ken Kennedy — an influential world leader of parallel computer field in 1994.

In the meantime, the world has witnessed the fantastically successful history of sequential computer systems based on so-called von Neumann model. The first von Neumann machine EDVAC [16] based on von Neumann architecture [39] was proposed in 1946 and soon found significant commercialization opportunities. Why does von Neumann architecture (the sequential computer architecture as it has been known) have such a robust history for over the past 60 plus years and continue to flourish in many ways into the new millennium? why does sequential approach continue more successfully than the parallel, given the advantages of the parallel approach?

One of such pillars is a robust sequential programming model whose foundation is based on the Turing machine model and the von Neumann architecture model that specifies an abstract machine architecture to efficiently support the Turing machine model. What should be the equivalent pillar to support parallel computation?

Unfortunately, there is no commonly accepted parallel Turing machine model and a corresponding parallel abstract architecture model for its realization. However, much good work on parallel models of computation exist, such as Leslie G. Valiant's work, proposing the Bulk Synchronous Parallelism (BSP) model as a bridging model of parallel computation between hardware and software [38], and David Culler's work on the LogP model as a practical model of parallel computation which focuses on performance characterization of parallel algorithms [6] — both served as significant milestones in the path searching for good models of parallel computation.

Most of the popular work on parallel models of computation have their basis on the concept of ***threads***. Threads are a seemingly straightforward adaption of the dominate sequential model of computation in concurrent systems. However, as Ed. Lee stated in his paper "The Problem with Threads", "*they discard the most essential and appealing properties of sequential computation: understandability, predictability and determinism.*" [33]

Consequently, in this paper, we choose to begin our work from a Turing machine model — in a path that is not actively pursued in prior research of parallel models of computation. We believe that there is a need to clearly identify the two corner stones of a parallel Turing machine model: a parallel program execution model (PXM) and an abstract architecture model. The sound properties of parallel programs should be specified in a clean and simple program execution model (PXM), *which is not based on threads*, while the realization of PXM should be the role of the design of an efficient abstract architecture model — and there may well be more than one design choice.

A summary of main contributions of this paper is outlined as follows:

- A survey reveals a disappointing status of the field of parallel Turing machine model studies.
- A Parallel Turing machine (short named PTM) model is proposed. The formulation of the proposed PTM consists of two interrelated parts: (a) a parallel

program execution model; (b) an associated abstract architecture model serving as a guideline to individual physical machine implementations.
- We highlight how our PTM addresses some of the weaknesses of the existing parallel Turing machine models. The program execution model of our PTM is based on the concept of ***codelets*** (not ***threads***), that preserve good properties, like determinacy, which are keys to the realization of modular (parallel) software construction principles by a parallel abstract architecture model.
- We conclude by presenting certain topics as future work on parallel Turing machine models.

The rest of this paper is organized as follows. In Sect. 2, we describe the existing parallel Turing machine model in detail and suggest possible areas for improvement. Section 3 presents our proposal: a parallel Turing machine model called PTM based on the concept of codelets and codelet graphs. A PTM consists of a codelet graph and a memory. The program of our PTM is represented by a codelet graph (CDG), and the corresponding abstract architecture model is called CAM (codelet abstract architecture). An example is included to illustrate how a parallel program is executed under our PTM. Section 4 briefly reviews related work. Section 5 gives the conclusion of our work and raises some open questions for future study.

2 Existing Work on Parallel Turing Machine — A Disappointing Status Report

2.1 Existing Parallel Turing Machine Proposals

In 1936, Alan Turing invented an idealized computing device [37], named Turing machine (TM). Dr. Turing's model for sequential program execution influenced the programming model of sequential computers. Hemmerling [24] first tried to propose a parallel computing Turing model in 1979. His model consists of a number of finite state machines working on a shared Turing tape.

Wiederman generalized Hemmerling's model [40] in an article published in 1984. Wiederman's parallel Turing machine has one infinite tape, as does a sequential Turing machine, but it may have multiple processing units (PUs), namely multiple read/write heads. Each processing unit, working in a similar fashion as a single sequential Turing machine, can read the input from the tape and write the output back to the tape independently. Figure 1a illustrates Wiederman's model.

The behavior of a "program", namely the corresponding FSM of a parallel Turing machine, is quite different from that of sequential Turing machine. Consider a state S of a FSM that has several successor states $\{S1, S2, \ldots\}$ all corresponding to the same input. Under a sequential Turing machine, the transition from S will non-deterministically select one of the states from the set $\{S1, S2, \ldots\}$ to be next state and make a transition accordingly. Under Wiederman's Turing machine model, state S makes transitions to a set of states $\{S1, S2, \ldots\}$ *simultaneously*, as shown in Fig. 1b.

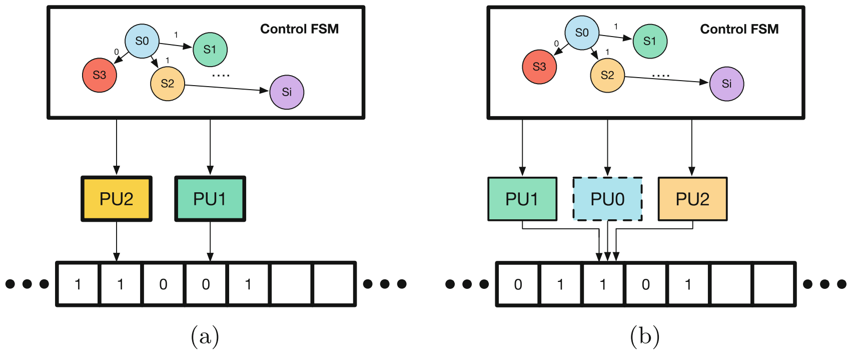

Fig. 1. (a) Wiederman's model: multiple processing units share a common tape; (b) Generating the new processing units: PU0 copies itself yielding PU1 and PU2, each with its own event in the shared FSM control. (Color figure online)

Wiederman's model starts with a single processing unit; as soon as the computation begins, when a running processing unit (PU) encounters any state with multiple succeeding states labelled with the same input, the FSM will begin to exploit the power of parallelism dynamically, i.e., the processing unit (PU) will produce copies of itself, so that there is one PU for each of the possible transitions, and carries out the operations.

Figure 1b and a illustrate this situation. In Fig. 1b, PU0 whose state is in $S0$ meets symbol '1'. Sharing the contol FSM, it provides for two processing units (PU1 and PU2) with events for state $S1$ and $S2$ respectively. After the multiplication, each of these processing units works as a separate Turing machine. However, all the new processing units in Wiederman's parallel Turing machine will still share the same FSM which is the same with the FSM before multiplying, as shown in Fig. 1a. The only difference between these PUs is that they are in different states of the FSM — the green PU1 is in state $S1$ and the brown PU2 is in state $S2$.

From our reading, in Wiederman's model, it appears to follow an assumption that all PUs run synchronously, which means they all finish a step in the same time. Thus additional wait states may be used to avoid serious conflicts (i.e., two memory operations happen concurrently and at least one of them is a write). However, if we allow the PUs to operate asynchronously, serious conflict, such as reordering writes to be after VS. before reads, may occur.

2.2 The Disappointing Status on Parallel Turing Machine Studies

There is little published work on parallel Turing machine. The few parallel Turing machine proposals we are aware of still use tapes as the model of storage in the corresponding architecture description. There seems to be little enthusiasm for or interest in searching for a commonly accepted parallel Turing machine model and corresponding abstract architecture model in the field of parallel computing. This is true even during the past 10+ years of the second spring of parallel computing.

Forty years have passed since Hemmerling's attempt to propose a parallel Turing machine model in 1979. In Computer Science, the field of *concurrency* — which covers both parallel computing and distributed computing — has advanced significantly since its infancy in 1960s and early 70s [31]. With the experience and lessons of the serious down turn of parallel computing in the 1990s, we also learned the importance of developing a standard, robust interface between software and hardware as a basis of machine independent parallel programming.

Today, any serious attempts on parallel Turing machine must have a solid model of memory beyond Turing's tape storage. As described in previous section on Wiederman's model, after the multiplication, those replicated processing units will operate in parallel but they still work on the same tape. So they will communicate with each other by reading and writing a shared memory location. However, concurrent access of the shared memory location may cause a *conflict*. The resolution of such memory conflicts must deal with the so-called memory consistency model of the underlied architecture model — a field pioneered by Leslie Lamport in 1978 [31] — at about the same time as the first parallel Turing machine work by Hemmerling was published.

3 A Parallel Turing Machine Model — Our Proposal

Like we have mentioned earlier, we choose to start our work on a parallel Turing machine model, instead of following the research of parallel models of computation. Section 3.1 gives a definition and description of concepts of codelets and codelet graphs. Section 3.2 demonstrates our proposed PTM, consisting of a PXM and an abstract architecture, while Sect. 3.3 shows an example of how the PTM works. Finally, in Sect. 3.4, we have a short discussion of determinacy property of our proposed PTM.

3.1 The Concept of Codelets and Codelet Graphs (CDGs)

Codelets. A codelet is a unit of computation that can be scheduled *atomically* for execution and it is the *principal scheduling quantum* (PSU) in codelet based execution model. A codelet may be made up of a group of (many) machine level instructions and organized with various control structures including loops. Once become ***enabled*** (i.e. ready to be scheduled for execution), a codelet may be scheduled to an ***execution unit*** (or EU) and be executed *non-preemptively* as an atomic PSU — keeping the EU usefully busy until its termination. The effectiveness of the non-preemptive model has been demonstrated through the mapping of a number of applications onto a many-core chip architecture with 160 cores [22].

Codelet Graphs. A program can be organized by a composition of multiple codelets into a graph called a ***codelet graph***. A CDG is a directed graph $G(V, E)$, where V is a set of nodes and E is a set of directed arcs. In a CDG, nodes in V may be connected by arcs in E. Every node in G denotes a codelet. An arc (V_1, V_2) in the CDG is a representation of a precedence relation between nodes V_1 and V_2. Such a relation may be due to a data dependence between codelet V_1 and V_2.

The concept of a codelet graph (CDG) has its origin in *dataflow graphs* [13]. In particular, it leverages the dataflow program graphs proposed in the McGill Dataflow Architecture Model [17]. Consequently, a unique feature of CDGs is that they employ the "***argument–fetching***" dataflow model proposed by Dennis and Gao [10,19].

Firing Rules. The execution model of a codelet graph is specified by its operational semantics — called ***firing rules***: A codelet that has received all the required events[1] — "signal tokens" on the corresponding input arcs of the codelet — will become *enabled*. Any enabled codelet can be scheduled for execution (or called "firing", a term used in dataflow models of computation). Firing of an enabled codelet will consume all the input events (remove the signal tokens), perform the *computation* as specified by the codelet and produce results, and generate output events.

3.2 The Parallel Turing Machine Model

Program execution model (PXM) of the PTM. In our model, a PTM consists of a codelet graph (CDG) and a *memory*. The CDG can execute a program, which has been described in Sect. 3.1. It serves as the function of a "program" just like the finite state machine (FSM) in sequential Turing machine. A memory consists of a set of locations. The content of each location is addressable by an addressing mechanism, and manipulated through a set of memory operations like load and store operations. For the purpose of this paper, the memory can be considered organized and operated as a RAM (random access memory). The memory organization itself is left to be defined by a specific abstract architecture model and will be discussed later. A *state* of a PTM consists of the state of the CDG and memory. At a particular time, a state of a CDG is its configuration at that time, while the state of memory is the content of the memory at that time. A configuration of CDG is the assignment of events on the arcs of the CDG. An initial state of the PTM is the initial configuraton of the CDG and the initial contents of the memory. The computation of a PTM is carried out by the *state transitions* defined as follows.

Assume the CDG is currently at a particular state. Based on the current configuration, there may be a set of codelets which are *enabled* (or in enable

[1] The term *token*, is from a familiar terminology of dataflow literature. In the rest of this paper, we use the term *events* to denote signal tokens (or event tokens) present on certain arcs.

state) and a subset of which is selected to fire. Firing of an enabled codelet consists of the following three steps: firstly, it consumes all the input events; then the processing unit will read the input data from memory, perform the computation as specified by the codelet; finally, it stores results into memory, and generates output events. When the computation (firing) of the codelet is completed, some new events will be produced, and placed on its corresponding output arcs according to the specific firing rule of it. Also, upon completion, the memory state will be updated. Upon the completion of the firing of all selected codelets, a new state of the PTM is reached — consisting of a new configuration and a new memory state. This will complete the state transition process.

An abstract architecture model of the PTM. Now, we present the abstract architecture model of our proposed PTM which we named as CAM — i.e., ***codelet abstract machine***, as shown in Fig. 2.

A CAM consists of (1) a collection of codelet processing units (CPUs) and memory units (MUs), (2) a collection of codelet scheduling units (CSUs), (3) a hierarchy of interconnection network (HIN). The CPUs and MUs can be grouped together by connecting through the HIN and form nodes of the CAM. A number of nodes can be connected into a cluster through the HIN. These clusters can be further connected and organized into higher-level hierarchy such as a hierarchy of trees, meshes, etc. The MUs in CAM are organized to share a single shared address space. Consequently, the design decision on the memory consistency model is critical. Our PTM and its CAM will follow a specific memory consistency model.

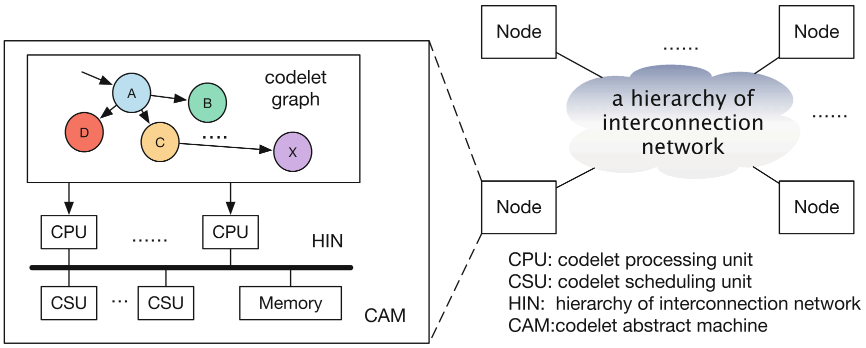
codelet graph
A
B
D
C
X
CPU
CPU
HIN
CSU
CSU
Memory
CAM
Node
Node
Node
Node
a hierarchy of interconnection network
CPU: codelet processing unit
CSU: codelet scheduling unit
HIN: hierarchy of interconnection network
CAM:codelet abstract machine

Fig. 2. CAM: codelet abstract machine

However, the details of the memory consistency model (MCM) and the structure of clusters connected by HIN are left as a design decision for any particular physical architecture that realizes the CAM. That is, we assume any physical architecture that implements the CAM will make the choice of a specific MCM and cluster structure that best apply to the underlying programming environment, hardware technology and other trade-offs so that a physical architecture and related machines may be built. Note that our CAM follows the stored-program principle as described by John von Neumann in the description of the EDVAC machine [39]. However, our CAM design may eliminate one-word-at-a-time between CPU and memory by permitting multiple data words be communicated concurrently between memory (made of multiple memory units) and CPUs.

3.3 An Example to Illustrate How the PTM Works

In Fig. 3 we illustrate a simple but informative CDG example of the proposed Parallel Turing machine. This "program" will first invert the tape contents and then change all the contents to '0'. If there are many execution units to do the work, which means each processing unit does the invert operation or changes the symbol to '0' at a specific memory location, the total excution time should be much less.

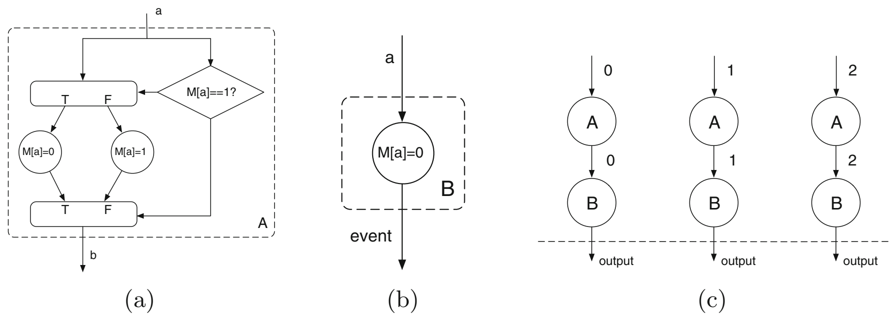

Fig. 3. A CDG example. (a) Codelet A: inverting the content of memory location a. (b) Codelet B: changing the content of memory location b to 0 (c) A non-conflict CDG

Codelet A (Fig. 3a) is a compound codelet which does the invert operation. It is defined as a conditional schema consisting of a decider, two codelets and a conditional merge. Codelet A is enabled once the input event (a) is available on the input arc. When fired, it will invert the content of memory location a. Also, an output event will be generated. Codelet B (Fig. 3b) is a simple codelet. It is enabled whenever the input event (a) arrives. When fired, it changes the content of memory location a to '0' and generates the output event.

Figure 3c illustrates a non-conflict CDG, which does exactly what we need and exploits as much parallelism as is presented in the problem to be solved. The three codelet As in Fig. 3c are first enabled by inputs and their outputs will carry the value of memory locations to be accessed and enable three codelet Bs. The numbers 0, 1 and 2 on codelet As' input arcs, act as the values of memory locations to be accessed. It means that when the input events arrive, the left codelet A, the middle codelet A and the right codelet A will invert the content of memory location 0, 1, 2 separately. So the contents of all the involved memory locations will be first inverted and then changed to ‘0’.

Figure 4 gives a detailed description about how the CDG is executed. A CDG could be executed on a CAM with arbitrary number of CPUs. Take Fig. 4 as an example. If there are more than three CPUs, three codelet As could be executed by any three of them. If there are fewer than three CPUs, the parallelism can not be fully exploited, only some of the enabled codelets (usually the same number as there are idle CPUs) are chosen by the CSUs and are executed first, then the CSUs continuously choose enabled codelets to execute until there are no more enabled codelets. Without losing generalization, we use a CAM with three CPUs to explain how this CDG is executed.

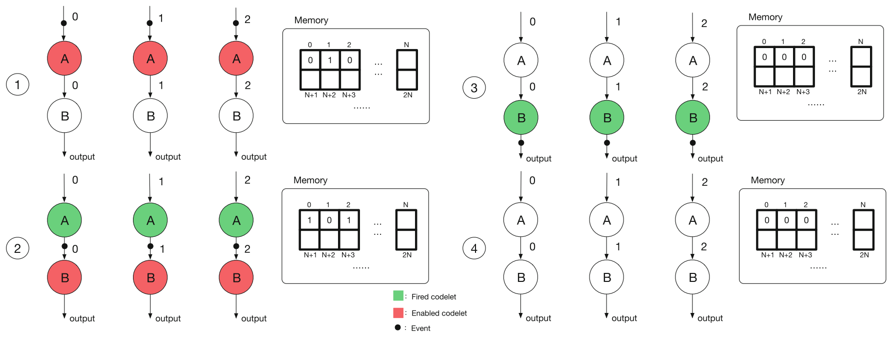

Fig. 4. The execution steps of non–conflict example

Figure 4 describes the detailed execution steps of the non-conflict CDG:

1. Step 1, the input events reach the corresponding input arcs of codelet As, thus all the codelet As are enabled.
2. Step 2, as there are three CPUs, the same number with enabled codelets, all these enabled codelet As are fired. They consume all the input events, invert the content of memory locations 0, 1 and 2, and then generate output events respectively. These output events further reach codelet Bs’ input arcs, thus all three codelet Bs are enabled.
3. Step 3, all the enabled codelet Bs are fired. They consume all the input events, change the content of memory locations 0, 1 and 2 to ’0’ and then generate output events respectively.
4. Step 4, all those enabled codelets are fired and no more enabled codelets left, now the computation is finished.

If there are enough units (say, six), we can change the CDG to let all these six codelets enabled at the same time to achieve significant speedup. Although this may cause conflicts when several codelets which have data dependence are scheduled at the same time, we could use a weak memory consistency model to avoid it.

From the previous examples, we can see that our proposed PTM has advantages over Wiederman's model: using event-driven CDG, we can illustrate a parallel algorithm more explicitly. What's more, since necessary data dependence is explicitly satisfied by the "events", our PTM could execute a CDG correctly regardless of the number of CPUs or whether they are synchronous or not. Meanwhile, using memory model to replace Turing tape, as well as CAM instead of read/write head, we make it more suitable to realize our PTM in modern parallel hardware design. Thus, we still leave enough design space for system architecture and physical architecture design. For example, we don't limit the design choices of the detailed memory consistency model and cluster structure, because these design choices may differ according to the specific application or hardware.

3.4 Determinacy Property of Our Proposed PTM

Jack Dennis has proposed a set of principles for modular software construction and described a parallel program execution model based on functional programming that satisfied these principles [9]. The target architecture — the Fresh Breeze architecture [12] — will ensure a correct and efficient realization of the program execution model.

Several principles of modular software construction, like Information-Hiding, Invariant Behavior, Secure Argument and Recursive Construction principle are associated with the concept of determinacy [14]. Consequently, the execution model of our proposed PTM laid the foundation to construct well-behaved codelet graphs which preserve the determinacy property. A more detailed discussion of the well-behaved property and how to derive a well-behaved codelet graph (CDG) from a set of well-structured construction rules have been outlined in [21], where the property of determinacy for such a CDG is also discussed.

4 Related Work

4.1 Parallel Turing Machine

There seems to be little enthusiasm for or interest in searching for a commonly accepted parallel Turing machine model. This is true even during the past 10+ years of the second spring of parallel computing.

In Sect. 2, we have already introduced the early work on parallel Turing machine models proposed by Hemmerling and Wiederman. Wiederman showed that his parallel Turing machine was neither in the "first" machine class [2], which is polynomial-time and linear-space equivalent to a sequential Turing machine, nor in the "second" machine class, which cannot be simulated by STM in polynomial time.

Since the publication of early work on parallel Turing machine models, some researchers have investigated different versions of these parallel Turing machine models. Ito [28] and Okinakaz [36] analyzed two and three-dimensional parallel Turing machine models. Ito [29] also proposed a four-dimensional parallel Turing machine model and analyzed its properties. However, these works focused on extending the dimension of parallel Turing machine model of Wiederman's work, but ignore its inherent weaknesses outlined at the end of Sect. 2.

4.2 Memory Consistency Models

The most commonly used memory consistency model is Leslie Lamport's sequential consistency (SC) model proposed in 1978 [31]. Since then, numerous work have been conducted in the past several decades trying to improve SC model — in particular to overcome its limitation in exploitation of parallelism. Several weak memory consistency models have been introduced, including weak consistency (WC, also called weak ordering or WO) [15] and release consistency (RC) [23] models.

Existing memory models and cache consistency protocols assume *memory coherence* property which requires that all processors observe the same order of write operations to the same location [23]. Gao and Sarkar have proposed a new memory model which does not rely on the memory coherence assumption, called Location Consistency (LC) [20]. They also described a new multiprocessor cache consistency protocol based on the LC memory model. The performance potential of LC-based cache protocols has been demonstrated through software-controlled cache implementation on some real world parallel architectures [5].

4.3 The Codelet Model

The codelet execution model [21] is a hybrid model that incorporates the advantages of macro-dataflow [18,27] and von Neumann model. The codelet execution model can be used to describe programs in massive parallel systems, including hierarchical or heterogeneous systems.

The work on codelet based program execution models has its root in early work of dataflow models at MIT [8] and elsewhere in 1960–70s. It was inspired by the MIT dynamic dataflow projects based on the tagged-token dataflow model [1] and the MIT CILK project [4] of Prof. Leiserson and his group. The codelet execution model extends traditional macro-dataflow models by adapting the "argument-fetching" dataflow model of Dennis and Gao [10]. The term "codelet" was chosen by Gao and his associates to describe the concepts presented earlier in Sect. 3.1. It derives from the concept of "fiber" proposed in early 1990s in EARTH project [26] which has been influenced strongly by the MIT Static Dataflow Architecture model [11]. As a result, we can employ a popular RISC architecture as a codelet execution unit (PU) [25].

The terminology of codelet under the context of this paper was first suggested by Gao, and appeared in a sequence of project notes in 2009–2010, that finally appeared in [21]. It has been adopted by a number of researchers and practitioners in the parallel computing field. For example, the work at MIT led by Jack Dennis — the Fresh Breeze project, the DART project at university of Delaware [21], and the SWift Adaptive Runtime Machine (SWARM) under the DOE Dynax project led by ETI [32]. The DOE Exascale TG Project led by Intel has been conducting research in OCR (Open Community Runtime) which is led by Rice University [34]. And the relation between OCR and the above codelet concept can be analysed from [43]. What also notable to this community is the recent R&D work pursued at DOE PNNL that has generated novel and promising results.

4.4 Work on Parallel Computation Models

In Sect. 1, we already include a discussion of related work on parallel computation models. For space reasons, we will not discuss these works further. However, we still wish to point out some seminal work following modular software engineering principles — Niklaus Wirth's programming language work of Pascal [41] and Modula [42], John McCarthy's work on LISP [35], and Jack Dennis's work of programming generality [7].

5 Conclusion and Future Work

This paper has outlined our proposal of a parallel Turing machine model called PTM. We hope that our work may encourage similar activities in studying parallel Turing machine model. We look forward to seeing significant impact of such studies which will eventually contribute to the success of parallel computing.

We suggest the following topics as future work. A simulator of our proposed PTM should be useful to show how the program execution model and the corresponding abstract architecture work. Two attributes of our PTM may by demonstrated through the simulation — practicability and generality. Since our PTM tries to establish a different model from the previous work, we will show how parallel computation can be effectively and productively represented, programed and efficiently computed under our PTM. Meanwhile, through simulation, we should be able to implement and evaluate extensions and revisions of our PTM. It may also provide a platform to evaluate other alternatives for abstract parallel architecture, such as those that utilize different memory models, or even an implementation that incorporates both shared memory and distributed memory models in the target parallel system.

Acknowledgements. The authors wish to show their profound gratitude to Prof. Jack B. Dennis at MIT whose pioneer work in computer systems and dataflow have greatly inspired this work. We are grateful to his selfless teaching to his students that served as a source of constant and profound inspiration. We also acknowledge support from University of Delaware and Tsinghua University which provided a wonderful environment for collaboration that led to the successful completion of this paper.

References

1. Arvind, K., Nikhil, R.S.: Executing a program on the MIT tagged-token dataflow architecture. IEEE Trans. Comput. **39**(3), 300–318 (1990)
2. van Boas, P.E.: Machine models and simulations. Handb. Theor. Comput. Sci. **A**, 1–66 (2014)
3. Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., et al.: The Illiac IV system. Proc. IEEE **60**(4), 369–388 (1972)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., et al.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)

5. Chen, C., Manzano, J.B., Gan, G., Gao, G.R., Sarkar, V.: A study of a software cache implementation of the OpenMP memory model for multicore and many-core architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 341–352. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15291-7_31
6. Culler, D.E., Karp, R.M., Patterson, D., et al.: LogP: a practical model of parallel computation. Commun. ACM **39**(11), 78–85 (1996)
7. Dennis, J.B.: Programming generality, parallelism and computer architecture. Inf. Process. **68**, 484–492 (1969)
8. Dennis, J.B.: First version of a data flow procedure language. In: Robinet, B. (ed.) Programming Symposium. LNCS, vol. 19, pp. 362–376. Springer, Heidelberg (1974). doi:10.1007/3-540-06859-7_145
9. Dennis, J.B.: A parallel program execution model supporting modular software construction. In: Proceedings of the 1997 Working Conference on Massively Parallel Programming Models, MPPM 1997, pp. 50–60. IEEE, Los Alamitos (1997)
10. Dennis, J.B., Gao, G.R.: An efficient pipelined dataflow processor architecture. In: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, SC 1988, pp. 368–373. IEEE Computer Society Press, Florida (1988)
11. Dennis, J.B., Misunas, D.P.: A preliminary architecture for a basic data-flow computer. In: Proceedings of the 2nd Annual Symposium on Computer Architecture, pp. 126–132. IEEE Press, New York (1975)
12. Dennis, J.B.: Fresh breeze: a multiprocessor chip architecture guided by modular programming principles. ACM SIGARCH Comput. Archit. News **31**(1), 7–15 (2003)
13. Dennis, J.B., Fosseen, J.B., Linderman, J.P.: Data flow schemas. In: Ershov, A., Nepomniaschy, V.A. (eds.) International Symposium on Theoretical Programming. LNCS, vol. 5, pp. 187–216. Springer, Heidelberg (1974). doi:10.1007/3-540-06720-5_15
14. Dennis, J.B., Van Horn, E.C.: Programming semantics for multiprogrammed computations. Commun. ACM **9**(3), 143–155 (1966)
15. Dubois, M., Scheurich, C., Briggs, F.: Memory access buffering in multiprocessors. ACM SIGARCH Comput. Architect. News **14**(2), 434–442 (1986)
16. Eckert Jr., J.P., Mauchly, J.W.: Automatic high-speed computing: a progress report on the EDVAC. Report of Work under Contract No. W-670-ORD-4926, Supplement (1945)
17. Gao, G.R.: An efficient hybrid dataflow architecture model. J. Parallel Distrib. Comput. **19**(4), 293–307 (1993)
18. Gao, G.R., Hum, H.H.J., Monti, J.-M.: Towards an efficient hybrid dataflow architecture model. In: Aarts, E.H.L., Leeuwen, J., Rem, M. (eds.) PARLE 1991. LNCS, vol. 505, pp. 355–371. Springer, Heidelberg (1991). doi:10.1007/BFb0035115
19. Gao, G.R., Tio, R., Hum, H.H.: Design of an efficient dataflow architecture without data flow. In: Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1988 (1988)
20. Gao, G.R., Sarkar, V.: Location consistency – a new memory model and cache consistency protocol. IEEE Trans. Comput. **49**(8), 798–813 (2000)
21. Gao, G.R., Suetterlein, J., Zuckerman, S.: Toward an execution model for extreme-scale systems-runnemede and beyond. CAPSL Technical Memo 104 (2011)
22. Garcia, E., Orozco, D., Gao, G.R.: Energy efficient tiling on a many-core architecture. In: Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores, MULTIPROG 2011; 6th International Conference on High-Performance and Embedded Architectures and Compilers, HiPEAC 2011, pp. 53–66, Heraklion (2011)

23. Gharachorloo, K., Lenoski, D., Laudon, J., et al.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of the 25th International Symposium on Computer Architecture, ISCA 1998, pp. 376–387. ACM, Barcelona (1998)
24. Hemmerling, A.: Systeme von Turing-Automaten und Zellularräume auf rahmbaren pseudomustermengen. Elektronische Informationsverarbeitung und Kybernetik **15**(1/2), 47–72 (1979)
25. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, San Francisco (2011)
26. Humy, H.H., Maquelin, O., Theobald, K.B., et al.: A design study of the EARTH multiprocessor. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 1995, pp. 59–68, Limassol (1995)
27. Iannucci, R.A.: Toward a dataflow/von Neumann hybrid architecture. ACM SIGARCH Comput. Architect. News **16**(2), 131–140 (1988)
28. Ito, T.: Synchronized alternation and parallelism for three-dimensional automata. Ph.D. thesis. University of Miyazaki (2008)
29. Ito, T., Sakamoto, M., Taniue, A., et al.: Parallel Turing machines on four-dimensional input tapes. Artif. Life Rob. **15**(2), 212–215 (2010)
30. Kennedy, K.: Is parallel computing dead? http://www.crpc.rice.edu/newsletters/oct94/director.html
31. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
32. Lauderdale, C., Khan, R.: Position paper: towards a codelet-based runtime for exascale computing. In: Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT 2012, pp. 21–26. ACM, London (2012)
33. Lee, E.: The problem with threads. Computer **39**(5), 33–42 (2006)
34. Mattson, T., Cledat, R., Budimlic, Z., et al.: OCR: the open community runtime interface version 1.1.0 (2015)
35. McCarthy, J.: LISP 1.5 programmer's manual (1965)
36. Okinaka, K., Inoue, K., Ito, A.: A note on hardware-bounded parallel Turing machines. In: Proceedings of the 2nd International Conference on Information, pp. 90–100, Beijing (2002)
37. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. In: Proceedings of the London Mathematical Society, pp. 230–265, London (1936)
38. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
39. Von Neumann, J., Godfrey, M.D.: First draft of a report on the EDVAC. IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993)
40. Wiedermann, J.: Parallel Turing machines. Research Report (1984)
41. Wirth, N.: The programming language Pascal. Acta Informatica **1**(1), 35–63 (1971)
42. Wirth, N.: Modula: a language for modular multiprogramming. Softw. Pract. Experience **7**(1), 3–35 (1977)
43. Zuckerman, S., Suetterlein, J., Knauerhase, R., et al.: Using a codelet program execution model for exascale machines: position paper. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT 2011, pp. 64–69. ACM, San Jose (2011)