A micro threading based concurrency model for parallel computing

Yang, Q.; Jesshope, C.R.; Fu, J.

DOI
10.1109/IPDPS.2011.323
Publication date
2011
Document Version
Accepted author manuscript
Published in
2011 IEEE IPDPS Workshops & PhD Forum (IPDPSW)

A micro threading based concurrency model for parallel computing

Qiang YANG, C.R.JESSHOPE, Jian FU
Institute of informatics
University of Amsterdam
The Netherlands
{ q.yang, c.r.jesshope, j.fu }@uva.nl

***Abstract*— The continuing launch of various multi-core processors backs parallel computing for gaining higher performance; however, it also exerts pressures on software developers concerning how to make full use of the multiplicity of resources to get the required benefits due to high coupling of parallel programs to specified hardware. In this paper, we propose SVP, a micro-threading based concurrency model, as an alternative to efficiently program on and manage multi-core, even many-core systems. SVP isolates programming from executing resources so that programs are compiled once and execute regardless of actual cores available, not only numbers, but also properties to achieve scalable speedup. Such lower coupling, better flexibility and portability in parallel applications are highly appreciated. To investigate SVP, tool chains and a fully functional software simulator of an SVP many-core chip have been developed for test and verification. Using this infrastructure, we are able to show how much we can gain from SVP and how speedup is scaled by running the same binary code of Game of Life on a scalable many-core platform.**

Keywords- parallel computing; concurrency model; multi core and multi threading;

I. INTRODUCTION

Off the shelf processors like server-side IBM POWER 7 and Victoria Falls from SUN, Intel's sandy bridge and AMD's Interlagos technology for personal computers, and GPUs from NVIDIA and ATI tell us that multi core technology has been mature and a commercial reality. It also promises parallel computing in the main stream since the end of frequency worship. Predictably, out of a need for higher performance, ongoing research and development from these giants is propelling computing towards an era with even larger parallel scale.

Nowadays, success of different multi-core architectures has proved that multithreading is a shortcut to better performance, but overcoming difficulties to make full use of these advantages to get what is required falls, to a great extent, on software developers and is notoriously hard. Performance not only relies on how concurrency should be exposed and managed, but also at what level of granularity is most suitable so that overheads of concurrency creation, synchronization and related communications are moderate. Additionally, a system expansion or reconfiguration often leads to new cycles of application optimization to improve performance for the new configuration. In this perspective, concurrent programs are constrained at quite low level of flexibility and reusability to achieve scalable speedup without labor-intensive work.

In order to solve these issues, SVP (Self-adaptive Virtual Processor), as the result of a decade of architecture research aims to give a holistic approach to programming many cores on chip. It subsumes new programming and execution models to isolate parallel programming from hardware characteristics and deploys concurrency to processing resources with a suitable granularity dynamically [1].

As a programming model, SVP addresses fine-grained parallelism with hierarchy to capture different levels of granularity with explicit dependencies. Both program development and compilation face abstract targets but guarantee deadlock freedom; thus programmers can focus on exploiting all potential concurrency without worrying about whether and how it will be supported by the specified hardware. Besides, mapping concurrency to an abstract target can avoid compatible issues.

As an execution model, SVP maps software threads to a certain platform and adjusts to hardware granularity at run time without any code transformation. Instructions are driven by control flow but threads are scheduled by data flow to tolerate long latency operations. Synchronizations of concurrent threads are always implicit and automatic without explicit management to avoid extra communication. Memory consistency is guaranteed internally to maintain determinism while forcing any access to critical sections sequential to eliminate the inevitable race conditions without locks or barriers.

The usability of SVP is strongly enhanced by tool chains developed by University of Amsterdam and other collaborators [2]. Beyond compatibility with C and C++, we propose a less generic programming language based on μTC [3] for SVP and an accompanying compiler based on GCC. The need for evaluation and verification gives birth to the Micro grid software simulator – a many-core architecture with RISC ISA implementing the SVP run-time system. With this support, performance evaluation can be carried out smoothly. And in this paper, we choose "Game of life" to demonstrate what the SVP benefits are.

The rest of paper is arranged as follows. More details of SVP model with tool chains will be given in section 2. Experiment, performance evaluation and analysis are located section 3, followed by future work and conclusion.

II. SVP AND THE SUPPORTING TOOL CHAIN

The initiation of SVP aims to solve two key issues of parallel programming, i.e. concurrency expression and resource mapping. In order to break their mutual influence, SVP integrates the micro threading based programming model and a run-time scheduler to take charge of each.

A. SVP Programming Model

The base of SVP programming model is called micro threading [4] originally designed for DRISC [5] (Dynamic RISC) — a data flow scheduling mechanism for RISC ISA.

In SVP programming model, concurrency is exposed by creating families of indexed threads that are statically homogenous but dynamically heterogeneous. Nesting and continuous creation is allowed to build a concurrency tree with levels of granularity from software component composition down to inner loops. A family creation has a cost of a few cycles and can be customized with desired thread numbers, thread function entry and then bound at run time to a collection of processing resources, a named place, which is the very abstract target adopted for programming. A place labeled with a unique id can directly be acquired by allocation from the place manager for a contractual amount of time like memory allocation. A place may be shared by different families by referencing the same identifier on creation.

A parent thread is allowed to execute concurrently and communicate with child threads it created asynchronously and a child family can collectively signal completion to its parent using synchronization. For the sake of locality and determinism, SVP applies constraints on interactions between parents and their children by explicit declaration of dependencies categorized into *globals* and *shareds*. *Globals* are written once by a parent and seen by all children but are read only. A write to a shared in a parent can only be seen and read by the first child thread, even it has created more, and the same action in a thread is only visible to its immediate successor in index sequence. The last thread's operations on *shareds* will be directly reflected on corresponding variables in parent's context that caused the initialization. So a unidirectional channel is built for communications and synchronizations between a parent and its children, as well as all threads of the same family. This pattern provides a well-defined sequential schedule and an abstract notion of locality to high level complier; it also guarantees deadlock freedom provided that there are sufficient resources.

To maintain memory consistency, only writes by the parent before creation are visible to all their subordinates; writes by child threads are visible to the parent only after family synchronization. Although it is loose and memory dependencies should be handled by explicit synchronizations, it can be efficiently achieved by bulk synchronization of pending memory requests on family creation and termination when implemented by hardware.

Additionally, exclusive places are provided for other unavoidable non-determinism like producer-consumer problems. The semantics of this approach are that families created at such a place must run sequentially so that accesses to any shared memory is exclusive. Therefore, concurrent operations on the same memory variables can be dispatched to the same exclusive place serving as a "secretary" e.g. in [6] without locks and semaphores. However, there is an additional constraint on memory consistency. Any thread running at an exclusive place must see all memory updated by any prior thread running at the same place. This is the only situation in which unrelated threads can share memory reliably.

B. SVP Run-time Scheduler and Simulator

The Micro Grid [7] is a software simulator that implements SVP many-core architecture by extending the Alpha ISA with instructions and mechanisms supporting the SVP programming model and a dynamic scheduler for concurrency mapping and management.

Basically, an SVP core is an in-order Alpha core with extension on memories and pipeline to support the programming model.

Since *globals* and *shareds* are mapped to basic data types (integer and float), the unidirectional channel is constructed on the i-structure based register files [8]. Registers in contexts of threads of the same family, no matter whether running on the same or a different core, are visible to each other so that communications through them are fast and efficient. Hence, two more ports are added to register files to provide this asynchronous communication.

The six-stage, in-order, multithreading pipeline switches between threads on either explicit tags inserted by the complier or on captured dependencies, instructions like branch or out of cache line boundary. These all hide long time latency operations. Obviously, the policy is fair and sufficient because the context switch is cheap and may be as fine as once per cycle.

Corresponding to the places in the SVP programming model, the Micro Grid provides different numbers of single cores, which are grouped into clusters connected by a dedicated control network on chip to build the framework of a many-core architecture. Fig. 1 shows a 128-core design with 1, 1, 2, 4, 8, 16, 32 and 64 cores respectively in the 8 places. Ring-based networks are used for intra-place control and communication. A separate COMA ring implements a two-level cache structure with a directory protocol.

Running on such a system, the SVP dynamic scheduler maps concurrency bound with a place in program to a cluster to perform either a local create on the first core of the cluster or a group create to distribute the required number of threads to all cores inside the cluster evenly and automatically. Any core can trigger a delegate creation and initiate subordinate families on any other cluster by passing a message through inter-cluster network. This scheduling tactic is performed on the basis of resource availability; meaning excessive concurrency exposed in programs will be waived spontaneously until sequential execution (only one single-thread per core is obtained) so that a suitable granularity will always match.

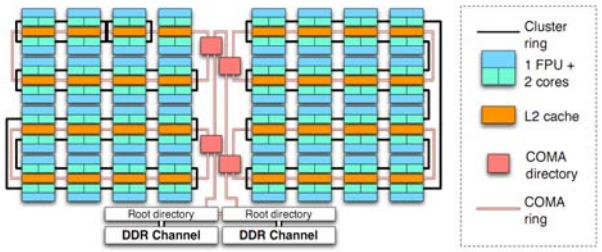


Figure 1. A propoesed design of SVP many-core architecture with 128 cores.

C. SVP Language and Compiler

SL (SVP Language) [9] is the proposed programming language for SVP. It captures semantics of SVP programming model as the sample code shown in Fig. 2 and is also compatible with plain C codes and platform neutral.

In SL, The place manager, SEP, is introduced to take charge of place allocation for client components or threads according to requirements and current usage. Explicit requests with different allocation policies will be processed in a strict sequential order.

The SL language also provides a monitoring framework and other built-in structures for run-time measurement and data collection to facilitate performance analysis and evaluation. An asynchronous monitoring thread can be triggered to execute with programs to collect cycle-accurate information and then generate statistical data in different forms for further processing. The general procedure is outlined in Fig. 3.

A GCC-based compiler generates binary code targeting the Micro grid using different ISAs from SL source code. Our partners are also developing high-level and more generic compilers from both standard (e.g. C, C++) and novel languages (e.g. SAC [10], S-NET [11]) targeting the tool chain we have developed for SVP and the FPGA implementation of a single core SVP chip based on the SPARC instruction set within the Apple Core Project of EU [12].

III. Experiments and Results

As is known, parallel computing prefers arithmetic intensive applications. Usually, such use cases may bring out its best to yield better performance. So does SVP [1]. However, a concurrency model should not be constrained within a finite field. If so, it will lose pervasiveness and become a specialized tool. Consequently, we carry out experiments on more complicated applications with quite few arithmetic operations to study the effect of SVP. Here, Game of Life is chosen as an example to demonstrate part of the work.

```
#include <stdio.h>

sl_def(foo, , sl_shparm(int, a))
{
   sl_setp(a, sl_getp(a) + 1);
   putchar('.');
}
sl_enddef

sl_def(t_main)
{
  sl_create(,, 0, 10,,,,
            foo, sl_sharg(int, x));
  sl_seta(x, 0);
  sl_sync();

  printf("\n%d\n", sl_geta(x));
}
sl_enddef
```

Figure 2. SL code example: t_main( ) creates 10 threads indexed from 0 to 9 at default place to print a dot respectively.

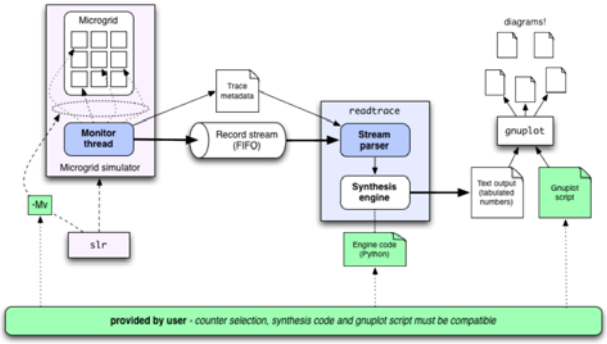


Figure 3. Monitoring thread: collects information of metrics and provides reports to gnuplot.

A. Game of Life in SL and Configurations

The popularity of Conway's Game of Life (GOL) may lie in the fact that cells' self evolution in an infinite two dimensional orthogonal square grid is only determined by initial inputs but can generate very complicated and unbelievable life patterns. People are not only interested in searching for inputs leading to such patterns, but algorithms to simulate the process rapidly and efficiently [13].

Although intrinsic dependencies between neighboring generations permeate the entire evolution procedure, yet it is still possible to exploit parallelism within the same generation for performance improvement. The major problem in implementing this algorithm is managing the infinite plane and ensuring an efficient processing of this sparse structure.

Among existing optimized schemes, we chose a relatively naïve one out of simplicity of parallelization. The structure of the three-stage procedure is shown in Fig. 4. It decomposes the infinite plane into chunks and maintains a list of active chunks to be processed in the next iteration. The algorithm is interesting in that it processes both chunks and list in parallel and also requires mutual exclusion.

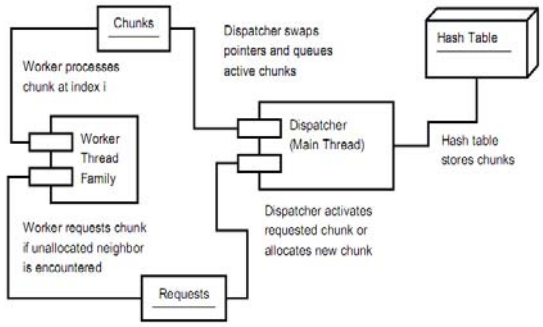


Figure 4. Algorithm adopted to implement GOL in SL. Three stages are used in processing each chunk, processing requests of new chunk allocation, updating chunk info and storing into the hash table.

Assuming that at the beginning of each iteration, there are M active chunks with a size of N×N, the performance cannot be better than O (MN^2) in sequential mode. Hence, exploiting concurrency within each stage has benefits but depends on the initial inputs, chunk size and granularities. For example, the trick of tuning N rests with the fact that a smaller one reduces useless cycles on less active chunks but increases memory and scheduling overheads, however, a larger one reduces references to neighboring blocks and requests to dispatcher but has a coarser granularity of active versus inactive regions.

As mentioned, GOL will be affected by initial inputs, chunk size and evolving generations. We select three interesting inputs with a distinct tendency of generating active chunks during iterations and each resulting unique life patterns. From Fig. 5, it can be seen that both *Herschel* and *Thunderbird* have summits and the former is oscillating around 4 after 150 iterations, the latter becomes stable after 250. *Rabbits* is quite another matter, as it fluctuates and keeps changing with no rules but the overall trend is increasing.

Restricted by candidate benchmarks, the lower bound of chunk size cannot be smaller than 10. Thus, we start from a single chunk of size 10×10 and scale to 100×100 with a step of 10.

In addition, another parameter called block size is also important. It determines the number of thread slots per family per core. Usually, it is only a software matter but here it plays a role of configuring the hardware granularity as it determines the number of simultaneous threads executed by a single core. We don't want to use the default value (i.e. hardware limited) but watch how this factor influences concurrency performance. Considering present commercial processors support upmost 8 threads per core, this parameter will be confined in range 1~16 accordingly.

B. *Profiles of Micro Grid Simulator*

We have discussed possible platform neutral factors influencing behavior and final performance of GOL. When running on the Micro grid, we still have to investigate hardware choices for better results.

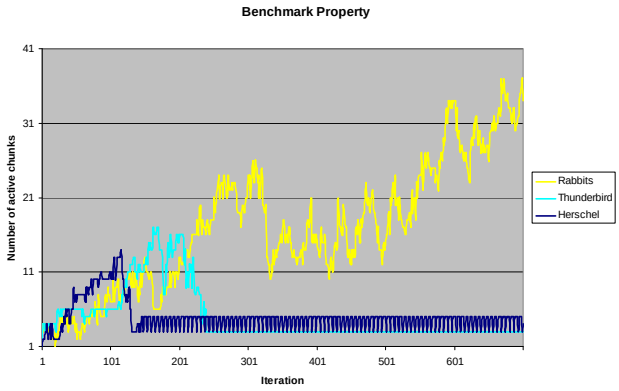


Figure 5. Benchmark properties viewed by the number of 10×10-size active chunks per iteration

Since the Micro Grid Simulator allows the targeting of different implementations of many-core architectures, we offer different configurations as profiles for flexibility. These profiles are used as a command line option and applied to the simulator at the stage of initialization. At present, the emphasis is put on place size and properties of memory hierarchy.

For GOL, c256_4 is more desirable. As implied in the name, this profile has 256 cores and 4 external DDR channels in total and a COMA memory. Compared with other configurations, more places are favorable for scalability; COMA is much more pragmatic than other memory models, and more DDR channels provide larger external memory bandwidth that may benefit the dynamic allocation of new chunks.

The general specifications of c256_4 are as follows.

- A 64-bit Alpha core with 1KB, 4-way set associative L1 I- and D-caches, 1024 integer registers and 512 floating-point registers, supporting a maximum of 32 families (including a reserved exclusive entry) and 256 threads. The clock rate is set at 1200MHz.
- A pipelined floating point unit is shared between two cores with 3, 8 and 10 cycles latency for add/mult, division and sqrt respectively.
- An on-chip COMA memory with two-levels of ring network and 4 DDR3-2400 1200MHz channels off chip with a bandwidth of 18.75GB/s per channel. At the top level there are 8 COMA directories each supporting rings of eight 32 KB, 4-way, set associative L2 caches with 64-Byte cache lines. Thus, 2MB on-chip L2 cache is available in total and offers a bandwidth of 75GB/s.
- 256 cores configured with an inter-place cross-bar network as 16 places comprising the following number of cores: {1,1,2,4,8,16,32,64,1,1,2,4,8,16,32,64} .

C. *Metrics and Evaluation*

What we are keen on illustrating is nothing more than speedup as the original intention of all work. It will be illustrated by comparing concurrent performance with the

cost of sequential execution on a single core with single thread slot. This can be achieved easily since the switch between sequential and parallel only depends on actual resources acquired; hence there is no need to change the code. Absolute performance in GFLOPS is not feasible because GOL rarely has FP operations. Computing resource utilization is another interesting metric and will be measured by pipeline efficiency.

While running on Micro Grid, function main () is designated to a single core place. An additional single core place is allocated acting as the exclusive place for potential races. GOL evolutions are dispatched to 1, 2, 4, 8, 16, 32 and 64 concurrent cores respectively to get the desired results.

Although each processing stage is parallelized, exclusive operations on shared memories exist and should be guaranteed. Therefore, it's difficult to quantify the exact proportion of parallel parts to the entire task, thus is not possible to estimate theoretical speedup.

Fig. 5 exhibits distinct properties of three different life patterns; whereas, all of them give quite the same results. Fig. 6 and Fig. 7 from *Thunderbirds* disclose the actual benefits from system expansion and reconfiguration separately by executing the same binary code of GOL.

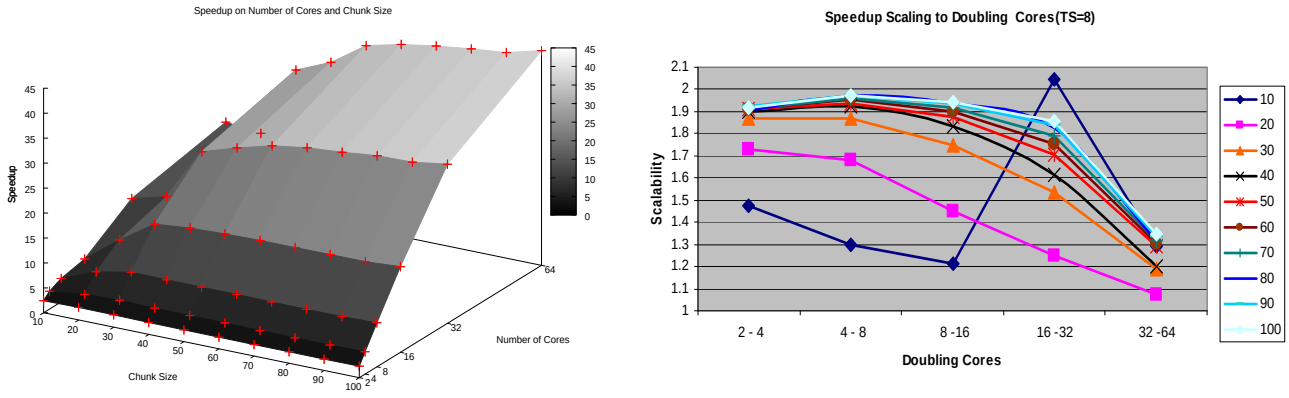
Fig. 6(a) shows the speedup from 8-thread slot cores with increasing numbers and chunk size. Viewing horizontally, when cores are fewer, e.g. 2, 4 and 8, speedup is almost insensitive to chunk size. For different chunk sizes, the speedup varies with available cores in varying degrees. Fig. 6(b) provides a precise illustration. It depicts the relative speedup for each doubling of the number of cores. Smaller chunk sizes always give descending curves from 4 onwards (a leap for chunk size 10 from 16-32 is rather an exception). Situations with larger sizes are better and from 4 to 16 cores, the scaling for each doubling is approximately between 1.9 and 2 but eventually this also drops for 32 and 64 cores. Mostly, this is resulted from the lack of concurrency while increasing computing resources. Costs of thread management, synchronization and communication offset the benefits from parallelization. From the curve of 100, it can be deduced that if chunk size is large enough, the speedup scalability will be much closer to 2 for long and delay the drop. However, all curves above 1 are accordant with Fig. 6(a) indicating speedup keeps increasing under any situations. Figures of other number of thread slots (i.e.1, 2, and 4) also give similar results with little difference. Putting all of this together, we can come to the conclusion that, for GOL, SVP does achieve a good scalable performance through providing more computing resources as long as sufficient concurrency is exposed.

Fig. 7 emphasizes the effect of varying the number of thread slots used but the same number of cores. As Fig. 6, we present absolute speedup and scalability from a 64-core place. In Fig. 7(a), speedup is increasing steadily from single thread slot onwards and reaches a summit at 8 but declines sharply at 16 for most cases. Such descent is reflected by curves in Fig. 7(b) falling from above 1 down to 0.3 and even lower between 4-8 and 8-16 for chunk size 30 and larger. It also shows that, for chunk size 10, speedup only changes at 1-2 and then keeps stable regardless of thread slot increase. The same phenomenon may occur to 20 from 8-16.

Explanation of this result should start from task allocation policy of SVP run-time scheduler. Out of fairness and workload balance, independent threads of a family will be distributed to all cores within a place averagely. Hence, difference of thread numbers among cores is 1 at most. Thread slot only sets the upper bound of the number of concurrent threads of a family in a core. Usually, a large value takes advantage of multithreading to hide latencies better, but it is not always the case. Let's take chunk size 30 as example. In a 64-core place, each core has to run at least 14 threads (15 for some 4 cores) to process a chunk. So thread slot functions from 1 to 14 and then becomes useless onwards. If it is small (i.e. 1, 2 or 4), benefits from thread scheduling is notable. Nevertheless, larger value may leads to troubles in that allowing too many threads to be created and running concurrently will aggravate resource competition, and lead to longer waiting time due to contention and rescheduling, memory saturation and bandwidth limitation. If these negatives gain the upper hand, speedup does decrease. The stability of line 10 from 2-4 is only because there are at most 2 threads per core and this number for chunk size 20 is 7. Defining a larger value for each is meaningless.

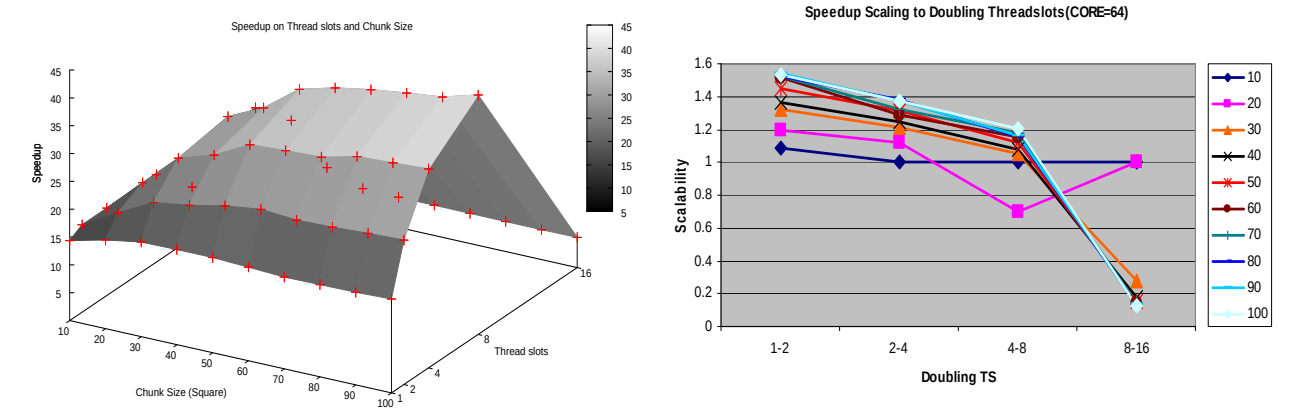
Compared with Fig. 6(b), the speedup scalability out of thread slot is not that good. As analyzed, there should be an optimal value as a balance point to generate acceptable result (like the initial value of larger chunk size in Fig. 7(b)). Anyhow, it indicates the passable performance of SVP on the change of thread slot as an example of core properties.

Beside speedup and scalability, we would like to study another closely related metric - system efficiency defined by the average pipeline utilization over all cores. This reflects the effectiveness of the SVP core's scheduling policy. Results from the change of thread slots and core numbers are presented in Fig. 8(a) and Fig. 8(b) as the average of concurrent cores because the other two, i.e. for function *main* () and the exclusive place, are used infrequently as they are always occupied with long latency operations and have no other threads to switch between. Fig. 8(a) shows the same effect of thread slots on system efficiency as it on speedup in Fig. 7(a). These results come out of similar reasons. The curved surface in Fig. 8(b) discloses the relationship among system efficiency, available concurrency and computing resources. More cores and smaller chunk size hinder multithreading from giving play to its advantages. It is the same conclusion we get from Fig. 6(a).



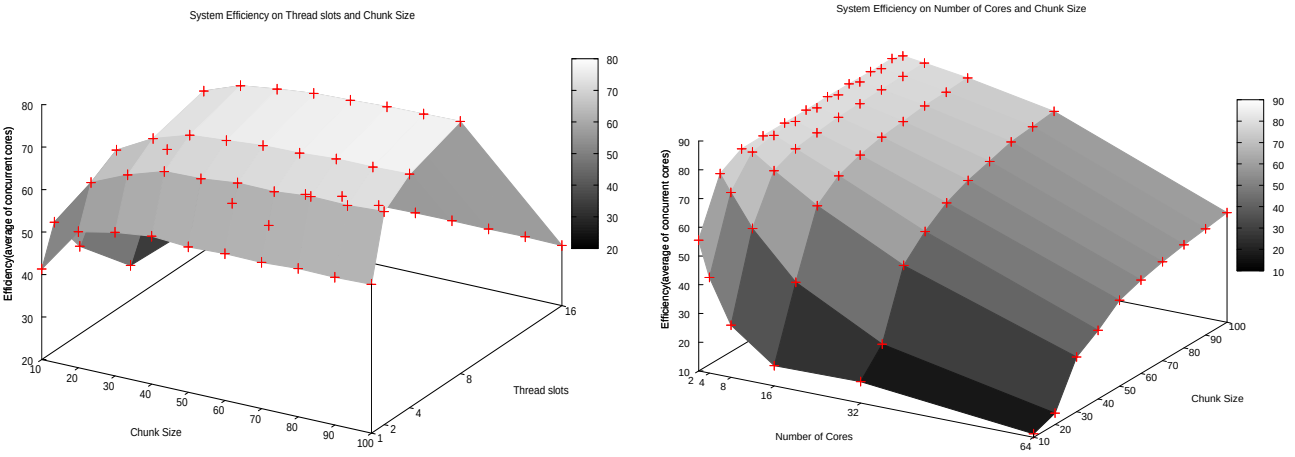
(a) (b)

Figure 6. Speedup resulting from system expansion - places with different number of cores : (a) absolute speedup (b) speedup scalabilty



(a) (b)

Figure 7. Speedup resulting from system reconfiguration - cores with different number of thread slots: (a) absolute speedup (b) speedup scalabilty



(a) (b)

Figure 8. System efficiency : (a) 8-core place (b) 8- thread slots cores

Briefly, System expansion and property alternation work on system efficiency similarly. Better speedup means higher efficiency, and verse visa.

IV. FUTURE WORK

SVP covers almost every important part of concurrent programming and control, which are evaluated here using the Micro Grid simulator. Using evaluations like this, we continue to improve on the previously published experiments on the Micro Grid. For example, currently the size of places is predefined (always as a power of two) and sometimes cannot meet the exact need of clients only because we adopt ring-based interconnection. A more flexible linear structure has been proposed to break this fixed topology constraint to provide configurable place sizes and also to allow virtualization of places. This helps find places of the required size for a given performance level and also allows for a measure of load balancing when many small families are created on a single large place. In SVP, concurrency creation is synchronized by default, but this may be too strict and hold creating thread back if the synchronization is not necessary [14]. Thus, support for asynchronous creation is also potentially useful. However, silicon implementation is most important and we have taken a step toward it. To cooperate with Intel, we'll work on SCC [15] to test and perfect SVP for many-core computing.

V. CONCLUSION

In this paper, we introduce the SVP concurrency model to decouple parallel programming from executing resources. Because of this property, applications need not be endlessly and tediously re-optimization due to changes of hardware but keeps scalable performance regardless of the number of resources used.

Supported by tool chains and simulator for SVP, we are able to make experiments for verification and Game of Life is one of them. As planed, we run the same binary code of GOL to see the benefits from SVP on scalable many-core architecture and the result is encouraging. No matter how many cores there are or what properties the core has, SVP will always yield a performance that is scalable to the configuration changes. However, more experiments and use cases are needed for further investigation. We will also transfer results from the software simulator to real hardware platforms based on the core's implementation in an FPGA.

REFERENCES

[1] C. Jesshope, M. Hicks, M. Lankamp, R. Poss, and L. Zhang, "Making multi-cores mainstream – from security to scalability," in Advances in Parallel Computing, vol. 18, IOS Press, 2010.

[2] D. Saougkos, D. Evgenidou, and G. Manis. Specifying, "loop transformations for C2µTC source-to-source compiler," in 14th Workshop on Compilers for Parallel Computing (CPC'09), 2009.

[3] C.R. Jesshope, "µTC - an intermediate language for programming chip multiproc-essors," in Asia-Pacific Computer Systems Architecture Conference, 4186 LNCS, 2006, pp. 147-160.

[4] C. R. Jesshope, "Microthreading a model for distributed instruction-level concur-rency ," Parallel processing Letters, vol.16, no. 2, 2006, pp. 209-228, ISSN: 0129-6264.

[5] A. Bolychevsky, C. R. Jesshope and V. B. Muchnick, "Dynamic scheduling in RISC architectures," IEE Trans. E, Computers and Digital Techniques, 143, 1996, pp. 309-317.

[6] E. W. Dijkstra, "Hierarchical ordering of sequential processes," Acta Informatica, vol. 1, no. 2, June 1971, pp. 115–138.

[7] Bousias, K., Guang, L., Jesshope, C., and Lankamp, "M. Implementation and Evaluation of a Microthread Architecture," Journal of Systems Architecture 55, 3 (2009), pp.149-161.

[8] Arvind, R. S. Nikhil, and K. K. Pingali, " I-structures: data structures for parallel computing," ACM Trans. Program. Lang. Syst., vol. 11, no. 4, 1989, pp. 598–632.

[9] Overview of SL language, http://notes.svp-home.org/sl2.html

[10] C. Grelck and S-B. Scholz., "SAC: A functional array language for efficient multithreaded execution," International Journal of Parallel Programming, vol. 34, no. 4, 2006, pp. 383-427.

[11] C. Grelck, S-B Scholz, and A. Shafarenko, "A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous compo-nents," Parallel Processing Letters, vol. 18, no. 1, 2008, pp.221- 237.

[12] The Apple Core Project, http://www.apple-core.info/

[13] Conway's Game of Life, http://en.wikipedia.org/wiki/Conway_Game_of_Life

[14] R.C.Poss, and C.R.Jesshope, "Lazy Reference Counting for the Micro grid," CGO'10 April, 2011, Chamonix, France,in press.

[15] Intel Corporation, Single-chip Cloud Computer, http://techresearch.intel.com/projectdetails.aspx?id=1