Molatomium: Parallel Programming Model in Practice

Motohiro Takayama, Ryuji Sakai, Nobuhiro Kato, Tomofumi Shimada
Toshiba Corporation

Abstract

Consumer electronics products are adopting multi-core processors. What is important for such adoption is to balance productivity and performance with high portability. To achieve this goal, we developed *Molatomium*, a programming model for the multi-core paradigm. It consists of a C-like coordinate language named *Mol* and a C/C++ part called *Atom*. In practice, our latest flagship digital TV, *CELL REGZA*^{TM}, uses Molatomium to parallelize its media processing applications. In this paper, we describe Molatomuim's programming model, language constructs and performance. Also, we discuss the pitfalls we faced during the development and the future plan.

1 Introduction

The multi-core paradigm shift is coming not only to personal computers but also to consumer electronics products such as TVs and smartphones. What makes it difficult to build software for such products is diversity. There are various types of processor architecture to be supported. Those can be homogeneous or heterogeneous multi-core, or have different cache hierarchies and unbalanced distances between cores. On the other hand, the tight development schedule places emphasis on software reusability. In addition, the short development cycle requires high productivity in writing efficient software. The tool for the development should have the ability to fully exploit the processing power. At the same time, it should ease writing parallel algorithms. In short, we need a way to develop efficient parallel programs with high portability with high productivity.

Molatomium is a practical approach to solve this problem. It has a coordinate language called *Mol* that ease describing parallel processing. Mol has a C-like syntax that many developers could feel familiar with. The parallelism is described declaratively in data-flow manner to make it hard for developers to put in concurrent bugs such as race condition and deadlock. Mol code is compiled to the portable byte code, which enables developers to port applications to other platforms without recompiling. *Atom* is the unit which is executed in parallel. Atom code is written in C/C++ and is compiled to native code. This allows developers to fully exploit the performance of target platforms. The balance between productivity of Mol and efficiency of Atom is the key of Molatomium. The runtime system is implemented as an autonomous distributed virtual machine. It takes care of the platform difference, schedules the concurrent execution of Atoms, and manages the dependencies among Atoms. Fig 1 shows the roles of Mol, Atom and the runtime.

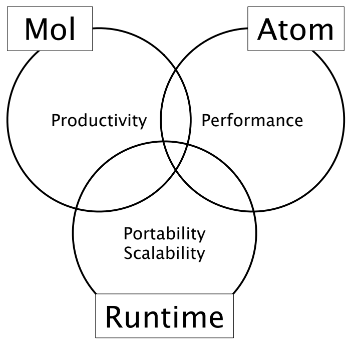


Figure 1: Constructs of Molatomium

Thinking and debugging a parallel program by text only is difficult. We developed a graphical development tool that consists of: an editor to support the designing process of parallel algorithms, a debugger to find out where the parallel program is wrong, and a visualizer to trace running parallel processes.

Molatomium is not just a research project but is used in the real world. It is shipped with our latest digital TV, *CELL REGZA*, which is powered by a Cell Broadband Engine^{TM} (Cell/B.E.)[1]. Molatomium is also used in our PCs, which is equipped with SpursEngine^{TM} (a variant of Cell/B.E.), to parallelize the applications such as image processing. These applications prove the practicality of Molatomium. Section 4 shows an experimental result about the scalability compared with that of other programming models.

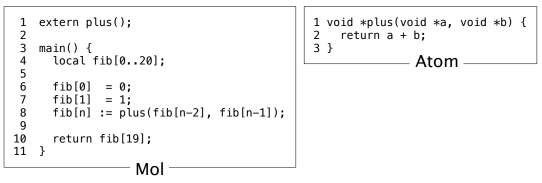
In developing software for consumer electronics products using Molatomium, we found another crucial factor that developers should pay attention to: *bandwidth*. We would like to share what we have learned in the development. In Sec 5 we discuss why the bandwidth matters and give an insight to manage it. Also, we explain a plan to expand Molatomium from multi-core platforms to distributed systems.

2 Programming Model

A Molatomium program is composed of two parts: Mol and Atom. The runtime system executes Atom codes concurrently according to dependencies expressed in Mol code. This separation makes it possible to construct a scalable yet efficient parallel program.

2.1 Mol

Mol is a coordinate language based on a data-flow model like Oz[6] and Cilk[1]. Its syntax is a C-like one. That reduces the migration cost that experienced C developers need to pay. Data dependency is expressed by connecting Atoms with array variables. Arrays are also used for memoization of computation. The runtime ensures that no simultaneous computation of the same array element occurs. Variables in Mol are treated as single-assigned data-flow variables.



```
 1  extern plus();
 2
 3  main() {
 4    local fib[0..20];
 5
 6    fib[0]  = 0;
 7    fib[1]  = 1;
 8    fib[n] := plus(fib[n-2], fib[n-1]);
 9
10    return fib[19];
11  }
```

Mol

```
1 void *plus(void *a, void *b) {
2   return a + b;
3 }
```

Atom

Figure 2: Fibonacci in Molatomium

Molatomium employs lazy evaluation to compute the data-flow variable. Fig 2 is an example of Mol and Atom code that computes Fibonacci numbers. Line 8 of the Mol code expresses the relationship among `fib[n]`, `fib[n-2]` and `fib[n-1]`. The use of ”`:=`” operator indicates that `fib[n]` is computed lazily. Its actual computation begins when line 10 is evaluated: it demands the value of `fib[19]` that leads to the actual computation of the evaluation of `fib[17]` and `fib[18]`, and continues the evaluation recursively.

Lazy evaluation lets developers just describe the data dependencies, not the order of computations. It enables the runtime to exploit the opportunities to parallelize the whole process, like out-of-order execution in hardware.

Mol code is compiled into platform independent byte code. The byte code semantics is simple load-store architecture, except that the execution of byte code is deferred if some of its operands have not computed yet. In particular, load instruction triggers some computation for its operands.

2.2 Atom

Atom is a serial code of parallel execution like kernels in OpenCL[4] and CUDA[5]. The Atom code written in C/C++ is compiled to platform native binary, and linked to the Molatomium runtime together with Mol byte code. In the Fibonacci example, `plus()` is a simple Atom code that returns the sum of its arguments.

Developers can write highly optimized code for the target platform, such as utilizing SIMD instructions.

Currently Mol is an untyped language and `(void *)` is used to pass all data between Mol and Atom. While Atom code should take care of the data type between distinctly different Atoms, it provides the developers with flexibility similar to C.

While developers of Atom should take care to interpret the data consistently by themselves, Atom code can be written for each platform. This provides developers room for optimizing the serial part by using platform specific knowledge and technique. Molatomium aims at the practical balance between portability and performance.

2.3 Runtime

Molatomium runtime is implemented as a virtual machine that resides in each core. It runs Mol byte code, invokes Atoms with maintaining dependencies, and manages the data flow. Fig 3 shows the states of cores at runtime. Current design of the runtime targets the shared-memory multi-core. The runtime stores its state and a global lock in a shared memory.

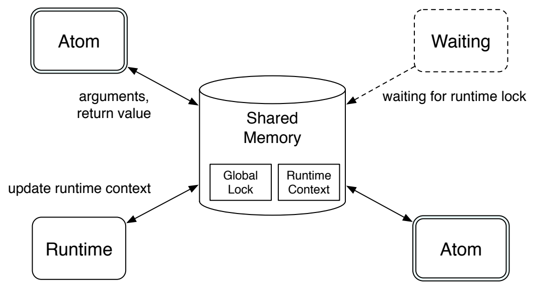


Figure 3: States of Each Core at Runtime

At most, one core can acquire the global lock to be an executor of Mol byte code. The core holding the global lock interprets Mol byte code sequentially until it encounters an executable Atom. Other cores run Atom code or wait for the global lock. Those core behaviors result in an autonomous runtime switching, and dynamic scheduling of Atoms. The absence of central scheduling core improves the scalability.

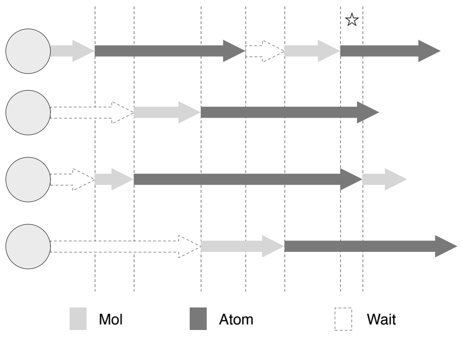


Figure 4: Core Activities

Fig 4 details the timeline of core activities. In this example, four cores are working on an application written in Molatomium. All cores are utilized for running Atoms in the period marked as ☆. Molatomium runtime is designed to maximize the period to achieve the good scalability.

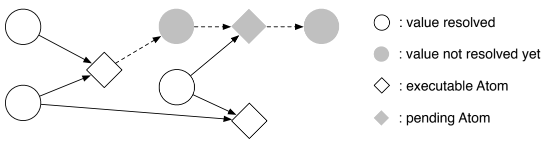


Figure 5: Data Flow Graph

Byte code execution builds a dependency graph. Fig 5 shows the graph which is constructed by the byte code execution. An executable Atom (white square) is one that all values of its arguments are computed (white circle). If there is an argument that is not computed yet (gray circle), the runtime creates a link for its arguments and return value pointer, enqueues its execution to the pending Atom collection (gray square), and then continues the byte code interpretation. After all arguments are computed, the runtime releases the global lock and starts executing the Atom code. This allows other core to become the next executor of Mol code by acquiring the global lock. When an Atom execution is finished, the core stores its return value into the return value pointer, and then waits for the global lock again.

3 Applications in the Real World

Molatomium has been employed as a basis to parallelize applications in consumer electronics products using Cell/B.E. and PCs with a SpursEngine.

```
extern run_edge();
extern run_ccr();
extern run_pocs_even(), extern run_pocs_odd();
extern get_loop_count();
extern get_frame(), put_frame();

main()
{
  frame(frame_no, input)
  {
     local loop;
     local edge[-1..9] outside(0);
     local pocs[-1..9] outside(0);
     local ccr[-1..9]  outside(0);

     loop     = get_loop_count();
     edge[i] := run_edge(input, i);
     ccr[i]  := run_ccr(edge[i-1], edge[i], edge[i+1], i);
     pocs[i] := (i%2==0)
       ? run_pocs_even(ccr[i-1], ccr[i], ccr[i+1], loop, i)
       : run_pocs_odd(pocs[i-1], ccr[i], pocs[i+1], loop, i);

     return &pocs;
  }

  sync for(i in [0..30]) {
    put_frame(frame(i, get_frame());
  }
}
```

Figure 6: Super Resolution in Molatomium

For instance, we have parallelized applications such as super resolution, frame interpolation, noise reduction, and various media codecs by Molatomium. Fig 6 shows the parallelized Super Resolution algorithm in Mol that is actually used in CELL REGZA.

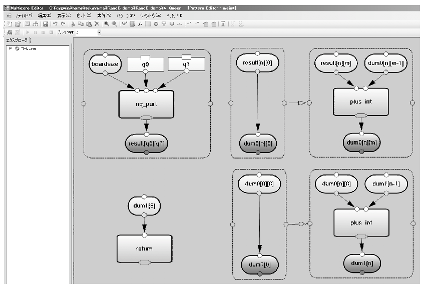


Figure 7: Editing N-Queen in IDE Editor

To enhance the development process further, we are developing a kind of IDE for Molatomium. Fig 7 displays the screenshot of the IDE. The IDE has an editor to create a Mol code intuitively by connecting Atom modules (rendered as rectangle), a graphical debugger to trace which thread alters the variable, a profiler to check which Atom is a bottleneck, and a 3D visualizer to observe how the parallel program is actually running concurrently.

In developing and parallelizing applications for CELL REGZA, first, we wrote and debugged the Mol code on x86 platforms, which are more flexible for tracing the problems. Then we moved on to the Cell/B.E. environment to link with highly optimized Atom code. This development process was less painful than was developing the application in Cell/B.E. platform from the beginning.

Migrating from x86 system to Cell/B.E. or SpursEngine platforms was painless. Developers just need to focus on Atom optimization rather than on parallelization. Therefore, Molatomium can help developers to raise productivity, and reduce the developing period by half compared with the traditional approach, and that shows the productivity of Molatomium.

4 Experimental Result

This section presents the experimental result of two benchmark programs. The main concern in the experiment is the scalability. We examined the scalability with respect to the platform difference and the productivity in parallelizing a serial application.

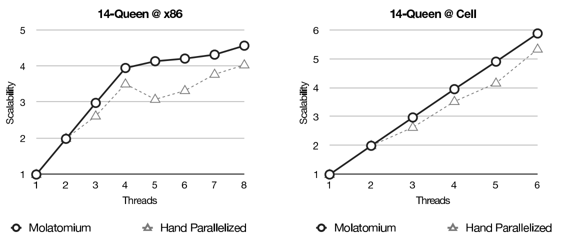


Figure 8: Evaluation of Scalability: 14-Queen

Fig 8 compares the parallel N-Queen program in Molatomium with a hand-parallelized one where N is 14. We evaluated it on a x86 system and Cell/B.E. platform. The x86 system consists of Intel® Core^{TM2} i7 (4 physical cores with hyper-threading) running Ubuntu Linux 8.04. We used Sony Playstation 3®[3] for the Cell/B.E. platform, which has 1 PPE and 7 SPEs running Fedora^{TM4} Core 9. Note that both programs on a x86 system scale up to 4 threads, because there are only 4 physical cores in this evaluation environment.

This result shows that how efficient the Molatomium's dynamic Atom scheduling is. The strategy of hand-parallelized scheduling is to assign Atoms statically to each core, while that of Molatomium runtime is to execute Atoms dynamically as soon as they are ready. It scales almost linearly up to 4 cores on x86 system, and fully utilizes all the cores on Cell/B.E. platforms.

Table 1 compares the scalability and the lines-of-code (LoC) needed to parallelize a serial code with those of other parallel programming methods. It runs *blackscholes* application included in the PARSEC[2] benchmark suite. We employ blackscholes because the PARSEC distribution bundles implementations of other parallelizing methods including Pthread, OpenMP, and Intel® Threading Building Blocks (TBB). The scalability is measured in the same environment above, using 4 cores. The result shows that Molatomium comes second on both aspects of the experiment. Molatomium has good balance that achieves good scalability with less code modifications.

Methods	Scalability	LoC
Pthread	3.92	100
OpenMP	3.04	40
TBB	3.9	125
Molatomium	3.91	75

Table 1: Comparing with other models about Scalability and LoC in blackscholes

5 Discussion

This section presents what we have learned in the development and the future plan of Molatomium.

5.1 Bandwidth

Though achieving good parallelism is a laborious task, it is not enough to optimize the parallel processing on multi-cores. There is another hidden factor: *bandwidth*.

We experienced a confusing issue in developing an application that processes lots of video data. We carefully balanced each task load and crafted data dependency paths so that no unnecessary waiting occurred, but could not achieve the required performance. What caused the slowdown was the huge traffic on the bus. The data transferred between cores consumed huge bandwidth and increased idle times on cores which is running Atoms whose arguments were ready.

Though we have a way to describe how much bandwidth is required, it has not been used effectively yet. The bandwidth reservation system can schedule the task dynamically by considering its bandwidth usage. If a task does not fully consume specified bandwidth, other tasks can utilize the rest of the bandwidth. That sounds nice, but leads developers to assign unnecessarily high bandwidth, because the requirement of the bandwidth changes in the whole development cycle. That results in contention and a decrease in performance.

The more the number of cores increases, the higher the traffic will be. Parallel programming models today should help in solving this issue.

We have preliminary implemented a dynamic granularity adjustment facility on Molatomium runtime (Fig

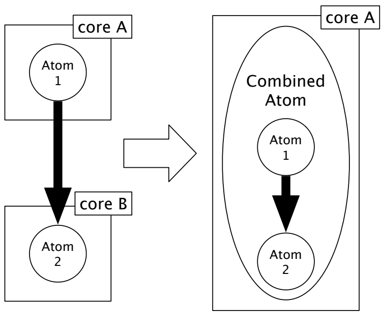


Figure 9: Dynamic Granularity Adjustment

9). When Atoms have dependencies and are assigned to distant cores, the transfer between cores consumes the bus bandwidth. This traffic can be avoided by combining those Atoms into one coarse-grained Atom so that the entire computations are executed on the same core. Thus it is recommended to make Atoms as fine-grained as possible so that the runtime can dynamically assemble them appropriately. To decompose coarse-grained Atoms dynamically is one of our tasks for the future.

What annoyed us is the invisibleness of bandwidth and its non-determinism. While we can see how good the cores are utilized with debuggers or hardware facilities like performance counters, there is no standard way to examine the bandwidth. We have started developing such system to analyze the bandwidth utilization that is available on different configurations of processor architectures and memory hierarchies.

5.2 Distributed System

Our target devices include discrete multi-processors as well. The programming model should be consistent between multi-cores and distributed multi-processors including heterogeneous configuration. Mol can be a coordinate language that glues Atoms running on such discrete processors. An Atom code can be optimized for each processor architecture and be combined into a fat binary.

In order to schedule effectively on such distributed systems, it is important to exploit locality. Some other parallel programming models express locality explicitly. For instance, X10[3] express locality as *places*. In Molatomium, we can use arrays to exploit locality between tasks implicitly. The runtime can assign a sequence of Atoms that computes the value of a single array to the cores that are close together. That would be a coarse grain task grouping, and be suitable for the situation in which the distance between cores are relatively long. Also, the runtime can assign the same core a series of Atoms which result in the value for an array element. This strategy is regarded as a fine grain task grouping, and as kind of data-parallel processing such as preformed by the GPU does.

To make Molatomium work on such distributed systems, we have to avoid using shared memory. One strategy is to adopt a distributed tuple-space model to share the runtime context, and that is another task for the future.

6 Conclusion

Consumer electronics industries are adopting multi-core processors. It is considered as promising of energy efficiency and high performance, but the programming model has not been matured yet. Molatomium eases developers from thinking about troublesome synchronization and helps them concentrate on developing the applications.

References

[1] R Blumofe, C Joerg, and B Kuszmaul. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices*, Jan 1995.

[2] Christian Bienia et al. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[3] Philippe Charles et al. X10: an object-oriented approach to non-uniform cluster computing. *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Oct 2005.

[4] Aaftab Munshi. Opencl: Parallel computing on the gpu and cpu., 2008.

[5] NVIDIA. Nvidia cuda compute unified device architecture programming guide, 2007.

[6] Gert Smolka. The oz programming model. In *Computer Science Today, Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

Notes

[1]Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and is used under license therefrom.

[2]Intel Core is a trademark of Intel Corporation in the U.S. and other countries.

[3]PlayStation 3 is a trademark or registered trademarks of Sony Computer Entertainment Inc. All rights reserved.

[4]Fedora and the Infinity design logo are trademarks of Red Hat, Inc.