# Porting a Domain Specific Language for Parallel Graph Analysis to the Many-Core Intel Single-Chip Cloud Computer

Alex Becherer, Dennis Brylow

August 1, 2012

**Abstract**

As CPU clock speeds rapidly approach practical limits, computer manufacturers are looking towards many-core architectures to continue to improve computational power. As a result, parallel applications are quickly developing to utilize this new computational paradigm. One particular mathematical area, graph analysis, can leverage parallel systems to greatly improve performance of computation on specific data structures that are well represented in graphs. Green-Marl, a domain specific language developed for easy and efficient graph analysis, provides an intuitive way for intermediate programmers to perform graph-analysis without having to understand the complexities of parallel programming. The Intel Single-Chip Cloud Computer (SCC) is a 48-core concept vehicle provided to accelerate research in parallel applications. A proposed system of porting Green-Marl to the SCC will explore the power of optimized parallel code running on a many-core architecture. This port should also give insights on the transition of mainstream computing to many-core systems, and contribute test data on analysis of graph based data-structures running in a unique and highly parallel environment.

## 1 Introduction

According to Moore's Law, computational speed should grow exponentially over time. In the first few decades of the mainstream production of CPUs, rapid growth of CPU clock speeds accounted for a significant portion of this growth. However, in recent years, CPU clock speeds have begun to hit practical maximums. Despite this, CPU manufacturers are still able to improve computational performance, often by simply increasing the number of cores on the CPU. After an already successful shift to multi-core machines and applications, corporations like Intel are beginning to investigate many-core architectures. Specifically, Intel has produced the Intel Single-Chip Cloud Computer, a 48-core concept vehicle for many-core applications research. This single-chip x86 architecture with built-in message passing hardware is an ideal platform to explore computing in

a many-core environment. Prior research on the SCC has produced SCC XINU, a port of the educational XINU operating system. This smaller, simpler operating system serves as a more configurable, lightweight platform than traditional Linux to help run parallel applications on the Intel SCC.

In a distinct, but not unrelated area of research, graph-theory is being used to analyze data that can be represented by graph-based data structures. Sophisticated parallel algorithms can provide drastic computational speed ups to graph-analysis but in turn, usually requires an extremely advanced knowledge of both graph-theory and parallel programming. To alleviate this requirement, the Green-Marl programming language was created by a group at Stanford. Green-Marl is a domain specific language designed exclusively for graph analysis with ease of use in mind. Green-Marl's syntax allows for average level programmers to write a simpler analysis algorithm that is translated into optimized parallel code in a target language. That optimized code can then be run in parallel, which allows for potential speed up without the difficulties of writing your own graph-analysis algorithm.

Due to the parallel nature of graph-analysis, a proposed system of porting the Green-Marl programming language to the Intel SCC will provide insight into parallel graph-analysis on a truly parallel architecture. A port of Green-Marl on top of SCC XINU will allow for easier low level configuration and potentially less overhead in applications. This system can help understand how traditional data structures can be better analyzed when represented by graphs, and generally improve knowledge about the transition to mainstream many-core systems. As a bonus, maintaining SCC XINU provides an educational tool for teaching parallelism at the operating system level. The remainder of this paper discusses in more detail the Intel SCC and the Green-Marl programming language, the difficulties and requirements of the port, and potential outcomes of a successful proposed system.

# 2 Prior Work

## 2.1 Intel Single-Chip Cloud Computer

To alleviate the decline in growth of CPU clock speeds, processor manufacturers are shifting towards multi-core and many-core architectures. These new many-core architectures feature dozens or hundreds of cores communicating over a Network on a Chip. Intel Labs has produced such a chip, known as the Intel Single-Chip Cloud Computer (SCC).[5] With this architecture, Intel hopes to "demonstrate a shared memory message-passing architecture" and to "design and explore the performance and power characteristic of an on-die 2D mesh fabric". The SCC is made up of 24 tiles, each with 2 GaussLake Pentium cores, making up a total of 48 cores. Each tile also has a router, 16 kB of shared memory (message passing buffer) for inter-tile communication, and 2 sets of L2 and L1 cache. The on tile routers use a packet-based routing mechanism that is transparent to the programmer. The SCC lacks cache coherency, which, for the

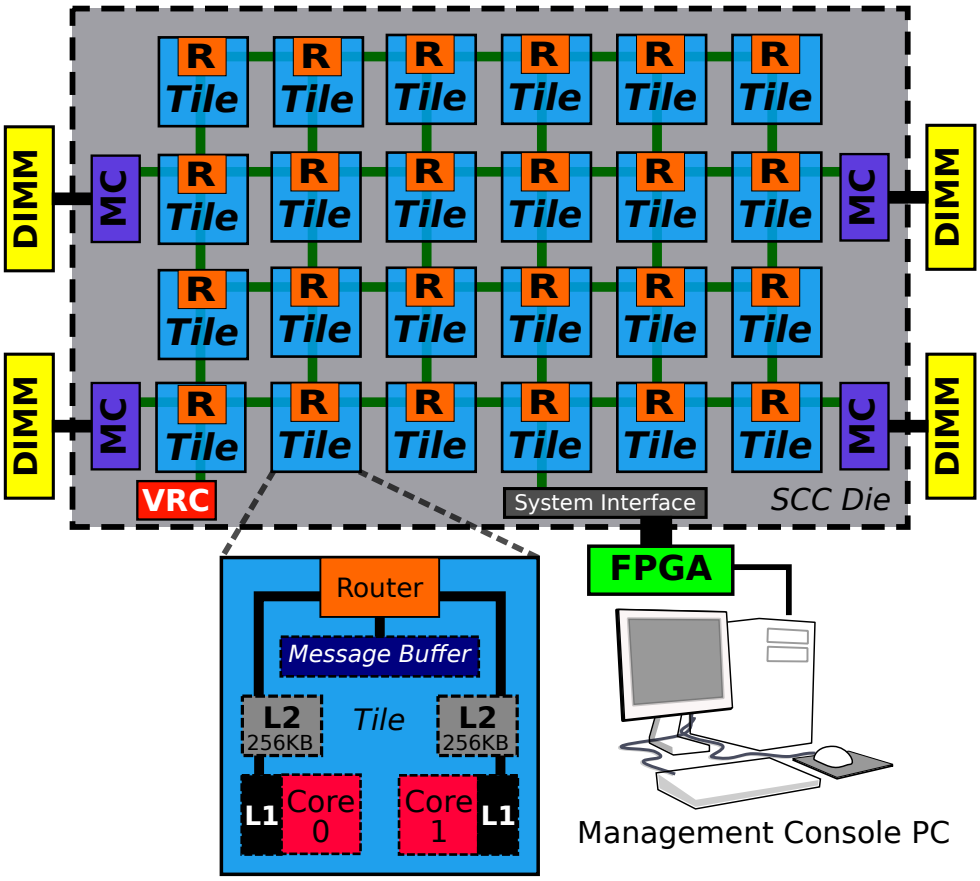


Figure 1: Block Diagram of the Intel Single-Chip Cloud Computer

purposes of this work, means that shared memory is not a realistic option for interprocess communication. Instead, the on tile message passing buffer offers a reasonably large and reasonably fast place to store messages from outside cores. In other words, the hardware present on the SCC makes it suitable for a message passing scheme, and not for shared memory.

The SCC features 4 memory controllers each connected to 4 of the outer tiles. These memory controllers connect to 4 off-die DIMM slots, configurable with up to a total of 64 GB of RAM. In addition, a voltage regulator controller allows the programmer to dynamically configure the frequency and voltage of each of the tiles. This feature allows users to study energy consumption, load balancing, and other power settings of the system and could be useful in our future work. The SCC is programmed through a management console PC (MCPC) which connects to the SCC through a PCI bus and a FPGA. SCC developers connect to the MCPC remotely, which in turn, connects to the SCC.

By default, the SCC boots with a version of Linux built for the SCC.[6] Each core loads its own image into off-chip private memory. In addition, Intel offers a baremetal image to get simple code running on the SCC. However, these are not the only options to programmers. Prior research has provided the SCC XINU operating system.

## 2.2 SCC XINU

XINU (standing for "XINU is not Unix") is a lightweight and simple Unix-like operating system developed by Douglas Comer at Purdue University.[2] It was designed and is still used as an educational operating system and has been ported to dozens of different architectures over the past few decades. The modern Embedded XINU port is a multitasking kernel with preemptive scheduling, inter-process communication and dynamic memory management. Despite all of these capabilities, XINU remains a very manageable size, comprising under 20,000 lines of code.

The most recent port of Embedded XINU is known XIPX (standing for "XIPX is parallel XINU") or SCC XINU. XIPX is a message-passing, thread-migrating operating system targeted for the Intel SCC.[11] Unlike several other popular operating systems for the SCC, XIPX offers the ability to run more than one concurrent parallel application. Unlike SCC Linux, XIPX is comprised of a very small kernel size, and relatively simple low level implementations that should offer much more customization. This lean size could also reduce overhead that a larger kernel might have. This simplicity has its drawbacks, however. XIPX is not POSIX compliant in many ways, most notably in thread structures. In addition, XIPX does not adhere to any message passing standards. Many parallel libraries require both of these standards, which means these shortfalls must be fixed in SCC XINU. Once these are implemented. SCC XINU should be much more ready to run a large pool of parallel based libraries.

## 2.3 Green-Marl

As the amount of data collected around the world every day becomes astonishingly large, developers are looking for new ways to process and manage their data. Graph-analysis is the process of drawing out further information from some given graph data-set. Many different industries including artificial intelligence, computational biology, and social networking are looking towards graph-analysis to extract information that may otherwise be very difficult to compute. Although graph-theory has existed for centuries, complex computational analysis that can take advantage of these non-traditional graph-based data structures is a fairly new science. Due to the rapid increase in multi and many-core systems and the inherent parallelism of graph-analysis, tying these two concepts together is highly desirable. However, writing code that analyzes these data structures efficiently requires a very advanced knowledge of both graph-theory and programming techniques. Adding parallelism to the equation makes the task much more difficult and cannot be expected of the average programmer, nor the body that is hoping to gain from graph-analysis.

To help alleviate this problem, the Pervasive Parallelism Laboratory at Stanford has created Green-Marl, a domain specific programming language for "easy and efficient graph analysis".[3] Green-Marl has a few central goals in mind. First, it looks to take advantage of the data-parallelism that is present in nearly all large graphs. By using parallelism, they can limit performance to memory

bandwidth instead of memory latency that would occur from single core processing. This parallelism can ideally take advantage of the user's multi-core CPU or even the machine's GPU, which can have hundreds of cores. Second, Green-Marl understands that implementing a graph algorithm correctly, efficiently, in parallel, and targeted for different environments is extremely difficult. It is not realistic to ask a single, average programmer to do all of these things.

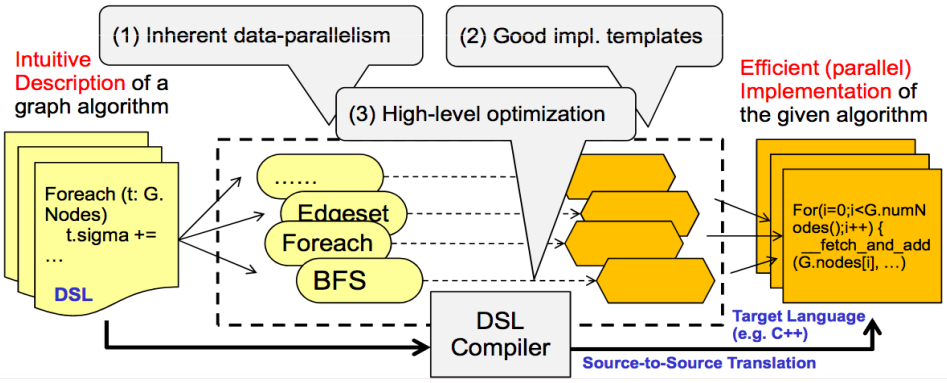


Figure 2: The Green-Marl compile process

The Green-Marl approach solves both of these problems. A user writes a graph algorithm intuitively and concisely in Green-Marl syntax. This code is then translated source-to-source to a target language (C, C++, CUDA or others) with high-level optimizations. The result is an efficient parallel algorithm written in another common language that can take advantage of a parallel execution environment if present. The Green-Marl DSL allows for a more productive coding experience, the portability of languages like C or C++, and optimized parallel performance. That being said, Green-Marl optimized code is still only as potent as the hardware it is run on. Running Green-Marl on a system that is highly parallel, such as the SCC, would be a very powerful endeavor.

# 3 Proposed Solution

A programming language designed specifically for parallel execution of graph analysis could clearly benefit from a highly parallel, single-chip environment like the Intel SCC. A port of the Green-Marl programming language to the SCC hopes to provide further insight on parallel algorithms running in parallel many-core architectures. Insight into this new computational paradigm could hopefully aid in answering questions about the transition into many-core architectures, and many of the other research areas that are still developing.

Specifically, a port of Green-Marl on SCC XINU would allow for more low-level analysis of an operating system running a parallel application that would be very difficult to uncover in a kernel as large as generic Linux. This port

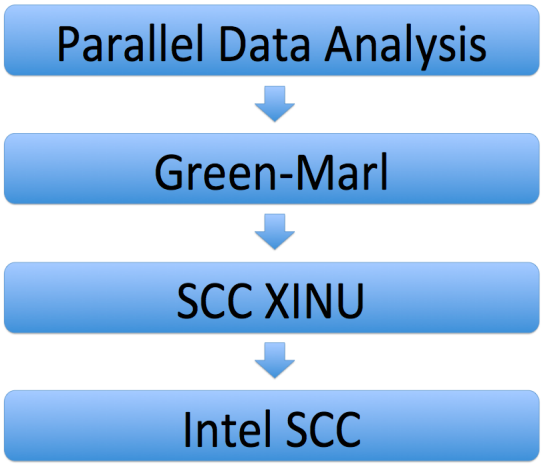


Figure 3: The Proposed System: Green-Marl running on SCC XINU

would also allow analysis of different message passing standards, and could provide a platform for testing these standards as they scale to a 48-core machine. In addition, because SCC XINU offers the ability to run multiple concurrent applications, we can reveal how applications like Green-Marl perform when the sequence of execution is not so clearly determined (because other applications may be using clock cycles, or interrupting other processes). Although this solution should reveal some very interesting ideas in parallel computing, porting of this language to both SCC Linux and SCC XINU does not come without its difficulties.

### 3.1 Green-Marl on SCC Linux

Running Green-Marl on SCC Linux appears to be not nearly as complex as the port to SCC XINU. There is only one major step involved in our current plan to port Green-Marl to SCC Linux: a working cross-compiler. The SCC processing cores are altered versions of the P54C Pentium design.[5] These Pentium cores are extremely old technology considering the hardware paradigm in which they are involved here. Compilers made in the days of these Pentium cores were not interested in message-passing standards, or any multi-core standards for that matter. As such, the cross-compiler offered by Intel for the SCC chip, does not compile applications that require a message-passing interface. Ideally, we would simply need to create a cross-compiler that is Open-MP compliant, and is able to target these Pentium P54C processors. Once that is done, the Green-Marl library should run smoothly on SCC Linux, just like it does on all other distributions of Linux that have a modern compiler. That being said, completing that step will most likely uncover a series of new problems, as most research steps do.

### 3.2 Green-Marl on SCC XINU

The port of Green-Marl to SCC XINU is a slightly more difficult process. There are two major shortcomings of SCC XINU that does not allow it to work out of the box for most parallel application libraries such as Green-Marl. First, the threads in SCC XINU are not POSIX compliant. Posix is a family of standards specified by the IEEE for maintaining compatibility between operating systems. Many applications rely on this standard as it is very common on many different operating systems, and allows for both applications and operating systems to be portable. To fix this problem, we could take one of two paths. We could alter the Green-Marl source so that it can work with the current thread model in SCC XINU or we can modify SCC XINU to become thread compliant. Although it is not clear which one is easier or faster to implement, the ladder allows SCC XINU to run many different parallel applications out of the box. This expandability is very desirable for future work.

The second major problem is that SCC XINU does not support the OpenMP API. OpenMP is a programming interface that supports multi-platform shared memory multiprocessing programming in C or C++. It includes compiler directives as well as library routines and environment variables. As the world shifts towards parallel computing, especially in non-embedded environments, most modern Linux distributions are offering OpenMP built-in. To include the OpenMP API on SCC XINU will require either a major change in the way SCC XINU actually passes messages or a software layer above the current message passing implementation that modifies the format of exchanged information. Fundamentally changing the message passing mechanism is probably the better solution as it would most likely retain a leaner kernel with less overhead. This choice will require a much deeper understanding of the system and will likely be the biggest hurdle in achieving the port.

## 4 Related Work

Although the SCC research community is very small in comparison to other computer science groups, there is some research that is very closely related to our work. A group at the Brandenburgische Technische Universitat is also working on Parallel Graph Algorithms for the SCC.[8]. Their work is more focused on the data sharing mechanisms, and involves an implementation of software level cache coherence to account for the fact that the SCC does not have hardware level cache coherence. They were able to show that shared memory could still be efficient despite the need for the overhead of software level cache consistency.

Another group at the University of Bayreuth was able to demonstrate how the general purpose programming language "Go" is able to take advantage of the built-in message passing hardware on the SCC to efficiently run Go in parallel.[7] Go uses "channels" that are similar to message passing but are more flexible in that they also serve to synchronize concurrent code. The group also showed, however, that their implementation failed to scale well when the number of

active channels began to overflow the limited message passing buffer.

Both of these related works are close relatives of our work on porting Green-Marl to SCC XINU. They both offer conceptual and implementation details that will be useful for endeavors in thread-migration, load-balancing, and other potential paths that our project may take in the future.

# 5 Conclusions

While a port of Green-Marl to SCC XINU is not a simple task, the resulting system would aid in the understanding of parallel applications running in parallel environments. The system would offer further insight into different message passing implementations and their effect on running several concurrent parallel applications. The port could help understand what type of industries stand to benefit from parallel graph analysis or if the computational speed up of parallel applications is even worth the trouble. As a bonus, the proposed system ensures a maintained version of SCC XINU, a very powerful tool for universities to teach parallelism at the operating system level.

Perhaps more importantly, this system offers a stepping stone for much broader research in the parallel computing paradigm. Using this system, researchers could explore anything from heat distribution and energy efficiency to creating mathematical models of deadlines in real time parallel operating systems. Allowing for these future endeavors and aiding in the understanding of parallel hardware, operating systems, and applications is what makes this proposed system a worthy project for the next few years.

# References


[1] D. Brylow, "An experimental laboratory environment for teaching embedded operating systems"," in *SIGCSE 2008: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, vol. 40. New York, NY: ACM, 2008, pp. 192–196.

[2] D. Comer, *Operating System Design: The XINU Approach*. Prentice Hall, 1984.

[3] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *ASLPOS'12*. ACM, March 2012.

[4] *Intel Architecture Software Developer's Manual, Volume 3: System Programming*, Intel Corporation, 1999.

[5] *SCC External Architecture Specification (EAS)*, Intel Corporation, November 2010.

[6] *The Scc Programmer's Guide*, Intel Corporation, January 2012.

[7] A. Prell and T. Rauber, "Go's concurrency constructs on the scc," in *Marc Symposium*, Toulouse, France, July 2012.

[8] R. Rotta, T. Prescher, J. Traue, and J. Nolte, "Data sharing mechanisms for parallel graph algorithms on the intel scc," in *Marc Symposium*, Toulouse, France, July 2012.

[9] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating Systems Concepts*, 8th ed. Hoboken, NJ: Wiley, 2009.

[10] M. Ziwisky and D. Brylow, "Baremichael: A minimalistic bare-metal framework for the intel scc," in *MARC Symposium*, Toulouse, France, July 2012.

[11] M. Ziwisky, "A message-passing, thread-migrating operating system for a many-core architecture lacking cache coherency," Master's thesis, Marquette University, 2012.