How Many Cores do We Need to Run a Parallel Workload: A Test Drive of the Intel SCC Platform?

Chen Liu\textsuperscript{a}, Pollawat Thanarungroj\textsuperscript{b}, Jean-Luc Gaudiot\textsuperscript{c}

\textsuperscript{a}Department of Electrical and Computer Engineering, Clarkson University, Potsdam, NY 13699, USA
\textsuperscript{b}Department of Electrical and Computer Engineering, Florida International University, Miami, FL 33174, USA
\textsuperscript{c}Department of Electrical Engineering and Computer Science, University of California, Irvine, CA 92697, USA

Abstract

As semiconductor manufacturing technology continues to improve, it is possible to integrate more and more transistors onto a single processor. Many-core processor design has resulted in part from the search to utilize this enormous transistor real estate. The Single-Chip Cloud Computer (SCC) is an experimental many-core processor created by Intel Labs. In this paper we present a study in which we analyze this innovative many-core system by running several workloads with distinctive parallelism characteristics. We investigate the effect on system performance by monitoring specific hardware performance counters. Then, we experiment on varying different hardware configuration parameters such as number of cores, clock frequency and voltage levels. We execute the chosen workloads and collect the timing, power consumption and energy consumption information on such a many-core research platform. Thus, we can comprehensively analyze the behavior and scalability of the Intel SCC system with the introduced workload in terms of performance and energy consumption. Our results show that the profiled parallel workload execution has a communication bottleneck on the Intel SCC system. Moreover, our results indicate that we should carefully choose the number of cores to execute different workloads in order to yield a balance between execution performance and energy efficiency for different applications.

Email addresses: cliu@clarkson.edu (Chen Liu), pthan001@fiu.edu (Pollawat Thanarungroj), gaudiot@uci.edu (Jean-Luc Gaudiot)

1. Introduction

Modern processor design needs to integrate massive number of transistors onto a single chip, following Moore’s Law [1]. However, it becomes more and more difficult to fully utilize the available transistor “real estate”. Frequently, the performance of a processor is limited by sequential program execution. A huge portion of the transistors is maintained in an idle state while waiting for a small portion of transistors to finish execution on a small portion of the sequential program in order to continue. Such problems could be solved by introducing parallelism into the hardware architecture so that more transistors could be utilized by processing the non-sequential portion of the program.

Many hardware techniques have been proposed and implemented to increase the parallelization of programs, such as Superscalar, Superpipelining, Simultaneous Multithreading (SMT) [2, 3], Chip Multiprocessors (CMP) [4] (also known as multi-core processors) and most recently, many-core processors. A multi-core processor is a single computing component with two or more independent actual central processing units (called “cores”), which are the units that read and execute program instructions [51]. If all the cores are identical, it is homogeneous architecture such as the Intel Core i7 [38]; Otherwise, it is heterogeneous architecture such as the IBM Cell Broadband Engine [36, 37].

There are several advantages of multi-core and many-core processors over single-core processors. First, both multi-core and many-core processors benefit from shorter wiring which minimizes the delay among cores instead of going off-chip. Second, the issue width and instruction window size cause a linear increase in the chip area for multi-core and many-core processors, whereas a quadratic increment is observed for single-core processor design if the same issue width/instruction window size is to be achieved [4]. Third, power consumption has been one of the critical concerns for computer system design due to the rapidly scaled-down integrated circuit: if we continue increasing the complexity of single-core processor design with increasing clock frequency, it will cause the power consumption to become prohibitive. This is the “Power Wall” problem [5, 6]. As for multi-core and many-core processors, power consumption increases linearly with the number of cores while
the clock frequency increases at a much slower pace. Fourth, the use of identical processing elements in homogeneous architectures reduces the overall hardware complexity and verification process, hence the entire development cycle.

The Single-Chip Cloud Computer (SCC) experimental processor [7] is a 48-core “concept vehicle” created by Intel Labs as a platform for many-core software research. The SCC system is the second-generation many-core processor design that has been successfully developed by Tera-Scale Computing Research Program [12]. The first generation was an 80-core chip able to deliver an astounding 1 teraflops performance. However, it was a prototype and not a complete design. On the contrary, SCC is a many-core processor with general-purpose Intel-Architecture (IA) cores integrated onto a single chip. There are built-in features such as high-performance power-efficient fabric, fine-grain power management and message-based programming support that come with SCC, which all intend to support more advanced research on many-core processor and parallel programming [15].

As we know, program behavior is quite dynamic. Some programs are extremely parallel; some of them are extremely sequential; others are somewhere in between. Even during the execution of a single program, the behavior can vary between different phases of parallelism. How we adapt many-core processor to such dynamic behavior is a great challenge. In this study, we have selected three representative workloads of various parallelism level and try to monitor their runtime behavior on SCC based on the execution time, average power consumption and overall energy consumption. SCC is a research platform that allows on-chip power management from user application directly. Thus, it is plausible to derive the power and energy consumption characteristics of the SCC when applying different frequency and voltage configurations to different cores during program execution. Our target is to explore how to utilize such many-core processor effectively based on different metric employed, either its being performance-driven, power-aware or energy-efficiency.

The rest of the paper is organized as follows. In Section 2 we start from briefly describing the SCC system as our research platform. Then we discuss the programming environment of SCC in Section 3. We also provide a detailed introduction of our voltage and frequency scaling setup on SCC in this section. In Section 4, we present our experimental results from varying the number of cores and varying frequency and voltage levels on different number of cores. Finally, conclusions and future works are presented in
Section 5.

2. INTEL SINGLE-CHIP CLOUD COMPUTER (SCC)

The purpose of designing the SCC chip is not to compete with any existing commercial processors. It is Intel’s research effort towards future many-core processor design. SCC contains an integration of 48 P54C silicon CPU cores. This specific core model was used in the Intel Pentium-series processors. One key concept when building this chip was to avoid overheating, so the Intel researchers went all the way back to original Pentium P54C processor to form the core in order to have the overall power consumption of the chip around 125W. That’s why it does not support Out-of-Order(OoO) execution, which would significantly increase the hardware complexity and make the power consumption beyond control.

Each core in the SCC chip has its own L1 and L2 caches. There are 16KB of L1 instruction cache, 16KB of L1 data cache and 256KB of unified L2 cache. The off-chip DRAMs, with a total size up to 64GB (only 32GB in our case), are separated into private memory units and shared memory units for each core. The memory sharing in DRAMs are managed through address lookup table (LUT). The cores operate at frequencies ranging between 100MHz to 800MHz. The SCC system consists of two separate computers, one computer houses the SCC chip with a custom-designed board called the Board Management Controller (BMC) and another one is a general system called the Management Console Personal Computer (MCPC). The communication between the two is done via PCIe interface as well as ethernet connection.

The MCPC hosts a Linux operating system which has the SCCKit installed. The SCCKit provides basic management over the SCC board such as reading or writing the SCC configuration registers, resetting a core and loading operating system images. The SCCKit also comes with both command-mode and GUI-mode applications for the user to config and monitor the SCC system. For example, one of the GUI applications is a performance meter that measures the utilization of each core together with the total power consumption of the entire SCC chip.

There is a shared directory where the MCPC and SCC can share their files. This directory can be used to store the application executables and transfer software extension packages in order to upgrade or modify the operating system installed on the cores. It is worth mentioning that each core
in the SCC can have its own native operating system installed. The SCC allows the user to load and unload the operating system image on any specific core. The operating system image on each core is responsible for running the commands and applications from the user. The user can implement a specific operating system to run customized applications on the SCC core.

Furthermore, high-speed embedded routers are embedded on the SCC chip. Every router is shared by two cores. The routers enable the cores to communicate with each other via message passing. The SCC designers opted not to implement any hardware cache coherency protocol, which is very expensive in terms of hardware overhead when the number of cores becomes sufficiently large. Instead, SCC chooses a software approach via their Message Passing Data Type (MPDT) protocol. By doing so, SCC alleviates the excessive overhead which would otherwise have incurred in order to implement the hardware cache coherency protocol, hence maintaining the scalability of the system. A small on-chip memory unit, called the “message passing buffer” (MPB), is implemented to accelerate the message passing process. Every two cores share a MPB of 16KB. It can be separated into two contiguous sections so that each core has its own 8KB of MPB. It can provide a means for fast message communication among the cores. The MPB can increase the message passing bandwidth to 1 GB/s whereas the router operates at a frequency of 2GHz. Fig.1 shows the SCC layout and its tile architecture.

RCCE [16] is an API library for the message passing programming model underlying SCC. An application running on SCC calls the RCCE library to manage message passing in the system. RCCE provides core numbering in SCC system, handles voltage level and operating frequency requests, and manages communication among cores. For example, an application on one core asks other cores to complete their current execution and hold for the next command by calling a function in RCCE. After that one core may call a function in RCCE again to collect all the computed results from other cores back to itself. In this way, the cores can maintain data coherency by not overwriting the computed result before it has been read. Unlike other general message passing APIs, RCCE is a compact API library that has been specifically developed for SCC, such that RCCE does not need any extra headers to communicate between cores. SCC also supports the commonly used parallel programming API library, Message Passing Interface (MPI) [18]. It has been proved in [17] that RCCE has lower communication overhead than using MPI. Message-based programming model is one possible way for cores
to communicate with each other when we scale up the number of cores on a single chip, in which case the hardware-supported cache coherence may not scale up. That’s why SCC has the on-chip network, on-tile router, and message-passing buffer to support the message passing programming model.

With the on-chip voltage regulator controller (VRC), SCC allows the user to dynamically adjust the voltage and frequency of the SCC cores. SCC can run the 48 cores at different frequency and voltage levels. Fig.2 shows the frequency and voltage domains, with the domains highlighted within thick lines. On one hand, the 48 cores can be separated into 6 voltage domains. Each voltage domain contains 8 cores and can have its own voltage level. On the other hand, the 48 cores are divided into 24 tiles with two cores per tile and each tile can have its own operating frequency. Please note there are two more voltage domains and four more frequency domains related to other on-chip fabrics. All of these features allow the SCC to perform fine-grain dynamic power management.

There are many debates regarding how to design the future many-core processors [52]. This paper is not intended to argue which approach is better than others. Many designing choices made when design the SCC chip may be subjective (not everybody would agree), but at least it is an effort and reflect the views from part of the community toward what future many-core processor would look like. The results we presented in this paper are all based a chip with MPI-style programming model and with no hardware supported
Figure 2: Frequency Domain and Voltage Domain on SCC Chip

cache coherence. It cannot be generalized towards other processors that do not fall into this design pattern.

3. Experiment Setup

In our experiments, we implemented our work on the SCC system with SCCKit version 1.4.0. MCPC is installed with 64-bit Ubuntu 10.04. The MCPC system supports basic C/C++ compilers such as gcc, g++, icc and icpc. Moreover, MCPC also supports a FORTRAN compiler ifort and an MKL compiler mkl. The executables are generated by MCPC and run on the SCC cores. Next we are going to introduce in detail the workloads we employed in this study.

3.1. Embarrassingly Parallel Workload

We first implemented an embarrassingly parallel workload program written in C that calculates the value of PI [15]. The overview of the PI program is shown in Fig.3 and the execution flow of the program is shown in Fig.4. We insert the PI computation algorithm from [30] into our program execution to calculate PI value. Originally, the algorithm was developed for parallel PI collective computation in MPI. We adapted the algorithm to efficiently use the RCCE library. The PI formula used in [30] is given in Equation 1. It calculates a tangent result from an integral. The integration of Equation 1 from interval 0 to 1 yields the value of PI. The PI algorithm discretizes
the integral into computable discrete terms. Hence, we can compute its integrand by summing up the discrete approximation tangent values within the integration interval. Fortunately, each discrete term can be computed independently from the others. Thus, we partition the entire discrete terms into a batch of iterative PI calculations. At each iteration, the computations are distributed evenly among the processing cores by assigning each processing core to compute one discrete tangent value per iteration.

\[
\pi = \int_0^1 \frac{4}{1 + x^2} dx \quad (1)
\]

We assign one of the cores the role of root core while all the others are computing cores. The root core is responsible for communication management and power measurement inside the SCC chip. During an iteration, the root core and each computing core calculate different portions of the PI value. PI values are calculated as a partial sum on each core independently. At the end of the execution, the root core will collectively combine all the partial PI results together to get the final value. In addition, it also calculates the error
Figure 4: PI program execution flow
compared to the actual PI value in 25 decimal places precision. Since it only takes a very short period of time to calculate the value of PI, we assigned a large number of iterations in an effort to extend the execution so that we can observe the chip behavior in a more stable fashion. In this experiment, we calculate the value of PI 100 million times in total. The PI program was compiled using the icc compiler version 8.1.038 with the optimization flag $O3$. The PI calculation itself can be computed in parallel, but there is additional overhead that cannot be parallelized such as the overhead from conditional loops, the timer, the SCC chip power consumption measurement, and the collection of the computed results.

3.2. Mixed Sequential and Parallel Workload

The second workload we implemented is a mixed sequential and parallel program written in C that calculates the product of two matrices [15], which we call matmul. The program execution overview of our matmul program is shown in Fig.5. To calculate the product of two matrices, we rewrite the computation algorithm from [31] to implement our message passing program model.

The algorithm was developed for parallel matrix multiplication collective
computation in MPI. During the initialization phase, the root core generates
two matrices to be computed. The first matrix is created by the addition
between the two coordinates of the matrix. The second matrix is created by
the multiplication of the two coordinates of the matrix. Then, the root core
will divide the first matrix into smaller portions based on the number of rows.
These portions of matrices and their offsets are being sent to other cores par-
ticipating the computation. This procedure also involves a scatter operation
that divides the entire data set into smaller pieces and sends them over to
different cores. Once the computing cores have finished their computation
on the multiplication of the matrix data received from the root core, they
will send back the computed portions with their offsets to be assembled at
the root core. This kind of operation is usually called “map reduce”, which
collects all the information from every participating cores back to the root
core. The initialization phase and the map-reduce phase must be performed
sequentially. The parallel computation happens when the participating cores
are computing the multiplication of the divided matrices. Due to memory
limitations, we use a matrix pair of 3000 rows by 150 columns and 150 rows
by 150 columns in size.

3.3. Embarrassingly Sequential Workload

Finally, we have implemented an embarrassingly sequential workload pro-
gram written in C that calculates a Cauchy sequence. A Cauchy sequence
is a sequence that converges to a limit. Sometimes it is used to evaluate the
stability of a system. Examples of Cauchy sequences are calculating roots,
ratios of Fibonacci numbers and trigonometric functions. The calculations
for these functions are sequential by nature because the value of the Nth term
depends on the value of the (N – 1)th term. We choose to implement the
square root calculation as our embarrassingly sequential workload program.
Here, we make use of the Babylonian method [32] to calculate the square
root as shown in Equation 2. The calculation could be done repeatedly until
the desired precision has been achieved.

\[ X_{n+1} = \frac{1}{2}(X_n + \frac{a}{X_n}), \text{ where } X_0 \text{ is an initial value} \quad (2) \]

To accommodate this sequential computation on the SCC platform, the
total number of iterations is equally divided among all the cores participating
in the calculation. The first portion of computation is performed on the first
core, then the first core uses the message passing ability to pass the computed
result onto the second core. The second core receives the computed result and continues to evaluate the square root for the same number of iterations as the first core did. After that, the result is passed onto the next core, until it reaches the last executing core, which returns the final result for the entire computation to the first core. We only measure the power consumption three times during the whole execution: after the initialization, after the first core sends its computed value to the second core, and after the first core receives the final result of the square root. We compute the square root 1 billion times iteratively. The computation is sequential by nature so as only one core will be executing at a time while other cores are idle. The redundant result handovers among the cores are the redundant message passing overhead, which will slow down the overall execution. An execution flow overview of our square root calculation program is shown in Fig.6.

The power management functions from RCCE library have been inserted into the PI program, matrix multiplication program and Cauchy program to adjust the frequency and voltage levels on the SCC cores. There are two sets of configurations in our experiments. First, we set the frequency and voltage level of every core to 800 MHz and 1.1V. Second, we set the frequency and
voltage level of every core to 400 MHz and 0.7V. These settings are applied to all the computing cores and the root core as well. By varying the number of cores, we were able to observe time, power and energy characteristics. Because not all cores were involved in the computation, there could be idle cores, the power consumption of which must be taken into consideration as well. Due to the fact that we cannot completely turn off the core even if it is in an idle state by using RCCE, we put all the idle cores into a low operational frequency of 200MHz with a supplying voltage of 0.7V. The operating frequency was fixed at the beginning of the execution. Thus, the program was executed at the same frequency and voltage until completion.

In order to observe the power and energy consumption characteristics, we added a software design model implemented by Intel Labs that allows us to obtain the reading of the instantaneous power consumption of the SCC chip, which was mingled into the program execution. Because the frequency and voltage scaling is one of the most commonly used techniques to lower energy consumption rate, we intentionally included the frequency and voltage scaling setup overhead by starting the timer right before setting the voltage and frequency levels. On the other hand, to counter the time spent to stop the execution of the program in order to get the power reading, we modified the original program such that it is now containing a RCCE timer which only counts when the program is running and stops counting when the program is fetching the instantaneous power measurement from the SCC chip. However, starting and stopping the timer is not enough to totally eliminate the overheads from such context switching. Other overheads include timer setups and power measurement done all along the execution, which cannot be easily removed. In this case, we sought to run our program long enough as to minimize the effect of those overheads on our results.

In terms of measuring power consumption, a single data point collected is normally not as precise enough. Therefore, the power measurement has been collected 100 times for every execution in the PI program, 5 times in the matrix multiplication program and 3 times in the square root operation program, which corresponds to the phase behavior of these three programs as illustrated in Figures 4,5,6 separately. The power measurement is computed from the product between the voltage supplied to the SCC chip and the total current flowed through the SCC chip. To perform a power consumption measurement from the SCC board, the root core must go through three steps. First, the root core needs to login to BMC which has the capability to read all the power sensors on the SCC board. Second, the measurement data cap-
tured from the sensors is processed and selectively written into a file within a shared directory among the 48 cores. Lastly, the root core is assigned to read and display measurement data from the file. The measurement algorithm is similar to the one done by the GUI performance meter provided by SCC. At the end of the program execution, the overall average power consumption was calculated from the mean of all power measurements performed within the execution. We executed the program starting from a single core all the way to the entire 48 cores.

4. Experimental Results

Our workload analysis is based on a version of HPCToolkit [19] that has been specifically developed for the SCC system. HPCToolkit was originally developed by Rice University to profile parallel workloads. It can perform an in-depth function call analysis of parallel programming workloads. HPCToolkit measures the performance of a program execution via performance counters available on the processor. For the case of SCC, performance counters reside in a hardware event monitoring register called “Control and Event Select Register” (CESR). The P54C core in the SCC system can support up to two independent hardware events monitoring at a time. We can monitor more than two hardware events by running the application repeatedly since the recorded profiling data are kept separately. The HPCToolkit can merge all the profiling data and represent them in a statistical table. The hardware events that we monitored are:

- Total instructions executed in the pipeline,
- L1 instruction cache misses,
- L1 data read misses and write misses,
- Pipeline stalled duration while waiting for a memory read.

By default, HPCToolkit always measures the total wall clock time of each individual function.

4.1. Workload Analysis

Fig. 7 shows the error percentage of the calculated PI value against the number of iterations. We compared our calculated PI value with an actual PI value of 25 decimal places. The graph shows a consistent linear decrement.
in error up to 1 million iterations. We can achieve a higher precision of PI value as we increase the number of iterations. Due to the fact that our PI computation has a limitation on decimal-point precision formula, we can only reach up to 12 decimal-point precision in this experiment.

Fig. 8 and Fig. 9 show the percentage of the total wall clock time spent on the top four active function calls of the PI program when executing using different number of cores. The highest percentage represents the time usage of either the heavy computation segment or the bottleneck segment of the PI program. All the performance counter data on different functions were measured exclusively, which means the child processes were not counted. Here, the top four function calls that consume most of the time during the execution are \texttt{main}, \texttt{f}, \texttt{RCCE\_wait\_until}, and \texttt{RC\_cache\_invalidate}, separately. Aside from calling all other functions, Function \texttt{main} also calculates the final sum of the PI value after collecting the results from other computing cores, as well as performs the power management function such as setting the voltage and frequency levels for each core on the SCC chip. All the timing and related communication functions are called by the \texttt{main} function as well. Function \texttt{f} is used to calculate partial PI value in each computing core. It was called repeatedly up to the total number of 100 million iterations. Function \texttt{RCCE\_wait\_until} is a waiting function required by several operations.
such as voltage and frequency level adjustment, execution ordering and sequencing, and the hold for PI computation completion on all the cores. It waits until flag in local MPB becomes set or unset. To avoid reading stale data from the cache instead of new flag value from the MPB, we should issue MPB cache invalidation before each read, including within the spin cycle. This has a byproduct of causing increased L1 cache miss rate, as we will see later. Function $RC_{cache\_invalidate}$ invalidates (not flush) all L1 cache lines that contain MPB data, which occurs from an update from another core. The cores need to wait for both computation and communication parameters to be updated.

At a very low number of cores, we can see majority of the execution time falls on the main function, which manages all other functions. A significant characteristic from the two figures is the non-linear increment on the time spent on the $RCCE\_wait\_until$ function, which increases abruptly after 16 cores and 32 cores in Fig. 8 and after 8 cores and 24 cores in Fig. 9, respectively. As the number of cores increases, the communication traffic among the cores is predicted to increase accordingly. $RCCE\_wait\_until$ is associated with the waiting time incurred during execution. This reveals the unbalanced increase in function call time as the $RCCE\_wait\_until$ function surpasses the main function after the 32-core point in Fig. 8 and 24-core point in Fig. 9. We would prefer to see more time spent on the targeting main function, rather than wasting the valuable wall clocks on the unfruitful $RCCE\_wait\_until$ and $RC_{cache\_invalidate}$ functions. These function calls slow down the computation process and yield longer waiting time in return. The high usage on $RCCE\_wait\_until$ results from the voltage and frequency level adjustment, handshaking and execution sequencing between the cores. Hence, the waiting function calls become one of the potential communication overhead, which could have a severe impact with higher number of cores.

By comparing Fig. 8 with Fig. 9, we can observe that there are also consequences of operating at a higher frequency. There is a higher wall clock count on the wait function when we execute the program at frequency of 800MHz. The decreasing wall clock count percentage in the $f$ function has been influenced by the $RCCE\_wait\_until$ function, in a similar fashion as the main function was affected. However, Function $f$ has a much lower total wall clock count, so the impact on Function $f$ is not so obvious. The increasing $RC_{cache\_invalidate}$ is conversely influenced by $RCCE\_wait\_until$. Recently, an improved version of RCCE, named iRCCE [20], has been released. The iRCCE is the non-blocking communication extension to the
Figure 8: Wall clock time percentage among the top four active function calls in PI program at 400MHz

Figure 9: Wall clock time percentage among the top four active function calls in PI program at 800MHz
RCCE communication library for SCC. It could reduce the communication overhead caused by \texttt{RCCE\_wait\_until} during program execution, but that is a subject of future work.

Both Fig.8 and Fig.9 show a non-scalable workload characteristic on the wall clock percentage, as the percentage of the \texttt{main} function continues to drop while the number of executing cores increases. Despite the fact that we are working with the same workload and the number of wall clocks required by the \texttt{main} function actually remain constant when we increase the number of cores, the number of total wall clocks required for the \texttt{RCCE\_wait\_until} function increases significantly with the number of cores, due to higher communication overhead. Another effect from increasing the number of cores is the growing number of wall clock spent on the \texttt{RC\_cache\_invalidate} function, which has the same growth direction with the \texttt{RCCE\_wait\_until} function, but at a much slower growth rate. The \texttt{RC\_cache\_invalidate} function has been affected by the effort to communicate and update new values among the cores. Fig.9 shows that when the frequency is high, the \texttt{RCCE\_wait\_until} function will influence other functions more aggressively.

Fig.10 and Fig.11 show the percentage of the total wall clock time spent on the top three active function calls of the matmul program when executing on different number of cores. The matmul program calculates the matrix multiplication operations directly under the \texttt{main} function. Similar to the PI program, the highest wall clock percentage represents the time usage of either the heavy computation segment or the bottleneck segment of the matmul program. Here the \texttt{RCCE\_wait\_until} function always takes more clock cycles than the \texttt{main} function. Even the number of clock cycles spent on both \texttt{main} and \texttt{RCCE\_wait\_until} functions are close to each other when we execute on only 2 cores, \texttt{RCCE\_wait\_until} function continuously expands its lead in total percentage over the \texttt{main} function when we increase the number of cores.

This suggests that the program spends more clock cycles on core communication part than computing the useful matrix multiplication. There are two major causes to this. First, the matrix initialization takes too much time as the initialization relies solely on the root core so that the other cores have to wait for the root core to completely create the two matrices in order to even begin their work. The second reason is the sequential communication pattern at the root core. The root core cannot broadcast to all the cores because different data elements of each divided matrix portion need to be sent to each individual core. The data has to be sent in a send/receive
Figure 10: Wall clock time percentage among the top three active function calls in matmul program at 400MHz

![Matmul - Total WallClock at 400MHz](image)

Figure 11: Wall clock time percentage among the top three active function calls in matmul program at 800MHz

![Matmul - Total WallClock at 800MHz](image)
pairing sequence from the root core to the other cores one by one. The root core now becomes the bottleneck for each communication handshake between the cores. The percentage spent on the \textit{RCCE\_wait\_until} function keeps increasing until it is saturated around 80 percent of the total wall clocks. The percentage spent on \textit{RC\_cache\_invalidate} increases slightly with the increase in that of \textit{RCCE\_wait\_until}. The communication overhead increases as we increase the operating frequency, as we can see that the percentage of \textit{RCCE\_wait\_until} at 800MHz is higher than the one at 400MHz.

Fig. 12 and Fig. 13 show the percentage of the total wall clock time spent on the top four active function calls of the Cauchy program when executing with different number of cores. Cauchy program is very efficient with single core execution because the entire program is obviously executed with very little data communication overhead. Function \textit{f} is the function call that calculates the actual value for each Cauchy sequence in each computing core. It was called repeatedly up to the total number of 1 billion iterations. Evidently, Cauchy’s \textit{RCCE\_wait\_until} function is the most lightweight function among the three workloads. It converges much faster than PI’s and matmul’s. This behavior is valid for the embarrassingly sequential workload because of its redundant data transfer between cores. It is very inefficient to compute a sequential workload like Cauchy execution on multiple cores. However, we can mimic such behavior to that of the hardware pipeline, which parallelizes the sequential instructions by dividing them into stages and overlapping the execution. We may optimize this sequential workload by executing the new sequential workload portion on the core that has just finished the previous portion of the workload [39]. This would help accelerate the process when there is more than one data-set for the same application. This optimization would help reducing the idle time for each core, improve the throughput, but it is not a cure for redundant data transfer.

Fig. 14 shows a comparison on the number of L1 instruction cache misses per kilo instruction (MPKI) between 400 and 800 MHz of the three different types of workload. As we add more cores into the PI execution, the number of L1 instruction cache MPKI increases consistently. This is because, with different number of cores, PI always executes with the same behavior throughout the entire execution, the only difference being the time and overhead affected from the \textit{RCCE\_reduce} function at the end of PI computation. PI has fewer L1 instruction cache MPKI when it executes at 400MHz. matmul has a low number of L1 instruction cache misses when it executes at 800MHz except the 2-core and 48-core cases. It suffers from a high number
Figure 12: Wall clock time percentage among the top four active function calls in Cauchy program at 400MHz

Figure 13: Wall clock time percentage among the top four active function calls in Cauchy program at 800MHz
of L1 instruction cache misses at 4 cores at 400MHz and 48 cores at 800MHz. Cauchy has a better number of L1 instruction cache MPKI at 400MHz except at 8-core and 24-core cases. Cauchy has a quite consistent L1 instruction cache MPKI because it shares a similarity with Pi in that each core executes the same amount of workload evenly. The average number of L1 instruction cache MPKI per core are decreasing continuously for the three workloads according to Fig.15, showing limited enhancement on instruction cache when we increase the number of cores.

The number of L1 data cache misses per kilo instructions could be used to predict the memory access pattern. It is normally used when the information on L2 cache access is limited. It can detect possible memory bottlenecks at execution time. Fig.16 shows the number of L1 data cache MPKI of the three workloads at different frequency levels. Without any doubt, Pi has very low L1 data cache MPKI compared to the other workloads because of its almost entirely parallel execution. The result suggests that running Pi at a lower frequency would influence the number of L1 data cache misses. It is likely that Pi also has low memory intensiveness due to the low L1 data cache MPKI. Nevertheless, the number of L1 data cache MPKI increases aggressively as the number of cores is increased. It is reflecting the dependencies on RCCE_recv and RCCE_barrier functions which are the main contributors to the L1 data cache MPKI. Tracing these two functions within the RCCE source code, we noticed that both of them directly use the RCCE_wait_until
and \texttt{RC\_cache\_invalidate} active function calls. The L1 data cache MPKI for matmul program at 800MHz frequency level is less than that at 400MHz frequency level, except for the 48-core case. The number of L1 data cache MPKI in matmul depends on the workload distribution. It is likely that the execution on 48 cores has a higher data cache MPKI because the overhead already dominates the main matrix multiplication workload. As for Cauchy, it results in higher L1 data cache MPKI on most of the cases when executing at 800MHz. Both matmul and Cauchy have a high L1 data cache MPKI, which is predicted to be a memory intensive workload. This could cause competitions on the memory bandwidth when running with other applications. Observing Figures 8 to 13, it can be noticed that executing at different frequencies (400MHz and 800MHz) results in different percentages of the total wall clock time spent on the top four active function calls, in particular the \texttt{RCCE\_wait\_until} and \texttt{RC\_cache\_invalidate} functions. Therefore, this can explain why in this case executing the same program at different clock frequencies would result in different cache miss rates. The L1 data cache MPKI per core decreases with increasing numbers of cores for matmul and Cauchy, as shown in Fig.17. The data transfer between the cores has the highest influence on L1 data cache MPKI. Fig.17 shows a higher L1 data cache MPKI per core as the number of cores increases for PI program.

The result for pipeline stalled duration while waiting for a memory read is an intriguing performance measurement since it could describe the mem-
Figure 16: Comparison on the number of L1 data cache misses per kilo instructions of the three workloads

Figure 17: Average number of L1 data cache MPKI per core for the three workloads
ory intensiveness of the system. It represents all the misses occurred because of memory reads, which also includes information on L2 cache and lower memory level misses. Fig.18 shows experimental results for pipeline stall duration while waiting for memory reads for the three workloads at 400MHz and 800MHz frequency levels. The PI program has few memory accesses and is less memory intensive than the matmul and Cauchy workloads. matmul does not perform well on the 2-core case because it executes sequentially and incurs more read misses than other cases. Overall, the duration of pipeline stalled while waiting for memory read increases with the number of cores, especially for the PI program. It seems that executing with a 400MHz frequency will cause less pipeline stall duration than executing at 800MHz. However, when taking the average, the pipeline stall duration while waiting for memory read per core actually changes little with the number of cores for all of the workloads, as shown in Fig.19. It is affected by the RCCE\_wait\_until and RCCE\_recv functions, which obviously add overhead to the computation and memory access. The results shown in the figure is also confirmed by Fig.16, in which the L1 data cache miss correctly predicted the memory intensiveness of the three workloads.

Our workload analysis shows that increasing the number of cores does not necessarily result in an absolute gain. It really depends on the nature of the workload. The communication overhead could become the bottleneck in the execution among the cores, incurred by the waiting function RCCE\_wait\_until. The speed of the execution is instead limited by the blocking communication behavior as well as cache misses. Nevertheless, for
the parallel workload, the SCC chip can maintain its scalability in term of computation time when we look into the overall execution time for each run. As we increase the number of executing cores, the amount of time required to finish the job indeed get reduced, as will be described next.

4.2. Energy Consumption Analysis

In this section, the experimental results are separated into two sets (800 MHz and 400 MHz). In our experiments, we recorded the total execution time and the average power consumption of the entire execution. Based on that, we calculate the total energy consumed by the SCC chip. Therefore, we have three different evaluation metrics for each execution on the SCC system: total elapsed time, average power consumption and total energy consumption.

First, Fig. 20 shows the relationship between the number of cores utilized for the execution and the time it takes to finish the execution for different frequencies with the PI, matmul, and Cauchy programs. For all three applications, the total execution time needed to finish the entire program execution at 800MHz is much less than the total execution time needed at 400MHz. Although the results from PI showed that we can reduce the execution time by increasing the number of cores within the 48 cores boundary, it does not mean that one should operate as many cores as possible for the execution. Because from the figure we can see that the performance of the PI program slowly converges at higher number of cores. This limitation is due to the on-chip communication overhead among the cores. Furthermore,
Figure 20: Comparison of the total execution time spent on the three workloads at 400 MHz and 800 MHz

The total execution time at 800MHz and 400MHz for the PI program are actually getting closer with each other as the number of cores increases. For example, program execution takes 0.64 seconds and 1.28 seconds with 16 cores at 800MHz and 400MHz respectively. The total time difference is 0.64 seconds. Whereas, it requires 0.36 seconds and 0.8 seconds to complete the program execution within 32 cores at 800MHz and 400MHz respectively. The total time difference is 0.44 seconds, which dropped down from the one of 16 cores by 0.2 seconds. This behavior is even more pronounced in the matmul program. With more than 15 cores executing, the total execution time starts increasing. The communication overhead now dominates the performance of the system and slows the matmul computation. It is not worth increasing the number of cores beyond 15 for matmul. Awkwardly, the Cauchy program does not exhibit any performance improvement as we increase the number of cores. The execution time saturates at 72.92 seconds at 800 MHz and 138.77 seconds at 400 MHz. The sequential portion of the execution means that there is a minimum amount of time to complete the workload due to its dependencies.

The execution time for an embarrassingly parallel program is naturally on a monotonically decreasing slope. This means that we can reduce the total execution time of an embarrassingly parallel workload program by increasing the number of cores: a higher number of executing cores yields less total execution time. This is not true in a mixed parallel and sequential workload program like matrix multiplication: when we increase the number of cores.
executing cores, the total execution time keeps reducing until it reaches the turning point, then it gradually increases. This phenomenon is the impact of communication overhead that increases with the number of cores. In such a mixed parallel and sequential workload program, it may be worth increasing the number of cores up to a certain point because there is a trade off between performance and communication overhead. For the matmul case, the system spends more time on the communication to exchange the matrix elements on each core when the number of cores is higher than 15 cores. The total execution time of an embarrassingly sequential workload is immune to the number of cores. However, we may improve the throughput of the embarrassingly sequential workload by running several of them in parallel in an overlapped fashion [39].

In Fig.21 we show the “scaling loss” of the PI program. The scaling loss is a calculation technique for evaluating the efficiency gain as we increase the number of cores [19]. The scaling loss computation is based on the time difference between two different numbers of executing cores, as shown in Equation 3:

\[
\text{ScalingLoss} = \frac{b \cdot \text{time}(b) - a \cdot \text{time}(a)}{a \cdot \text{time}(a)} \times 100\% 
\]

For example, Setup A executes the program with two cores which completes in 3 seconds, while Setup B executes the same program with four cores which completes in 2 seconds, then the scaling loss is \((4 \times 2 - 2 \times 3)/(2 \times 3)\) or 33.3%. This metric could be used to describe the scalability of the program on the SCC chip. It provides us with the information on the additional amount of time consumed when the number of cores is increased. The computation simply sums up the total time spent by each core to a single unweighed execution time. A higher percentage in the scaling loss computation indicates losses occurred by the increased number of cores. It is not worth increasing the number of cores when the scaling loss is greater than 100%, which means the execution entirely loses its performance when more cores are added as it spends more time to process the overhead rather than to process the original job. Our calculation compares the scaling loss of the current number of cores with the single core execution case as the baseline. The scaling losses for both 400MHz and 800MHz cases increase almost linearly with low number of cores, whereas it “oscillates” and becomes difficult to predict with high number of executing cores, but still on an overall upwards fashion. Another observation is that the PI execution at a higher frequency always yields a
higher scaling loss in this experiment. But in either case it is still less than 60%, which means it is worth increasing the number of cores for such parallel workload.

Fig.22 shows the calculated scaling loss for the matmul program. Firstly, compared to the PI program, the scaling loss here is much higher. As we can see, the scaling loss of the PI program never exceeded 60%, while matmul has a huge scaling loss with increasing numbers of cores which reaches almost 900% with 48 cores. This high scaling loss indicates not to perform the computation of matmul with a high number of cores because this would simply cause more overhead instead of useful computation. Secondly, the matmul program has a higher scaling loss when it is executed at a higher frequency. However, the effect of the scaling loss due to the frequency change in matmul is small. Thirdly, it is not worth increasing the number of cores after 15 cores because the scaling loss becomes larger than 100%, which coincides with our performance observation from Fig.20.

Fig.23 shows the calculated scaling loss of the Cauchy program, which is the worst among the three workloads studied. This is because the execution time cannot be improved due to its embarrassingly sequential nature. At 400MHz, the scaling loss increases linearly by 95% for every core added into the computation; while the scaling loss oscillates for 800MHz case, but still follows an increase trend-line.

Fig.24 shows the relationship between the numbers of cores utilized and the average power consumption at different frequencies. As we can see,
Figure 22: matmul scaling loss at 400MHz and 800MHz

Figure 23: Cauchy scaling loss at 400 Mhz and 800MHz
the frequency and voltage configuration has a huge impact on the power consumption. There is a dramatic increase in power consumption for the 800MHz and 1.1V setup as we increase the number of cores. Idle voltage domains were turned to a lower supply voltage level during the execution so that we can achieve a lower power consumption level. There is some specific number of cores that incurred a jump in term of the average power consumption. This jump is caused by turning on another voltage domain. The voltage domain are turned on at Core 4, 8, 24, 28 and 32 specifically based on our core numbering sequence. Fig.24 shows that the power consumption grows nonlinearly when we increase the frequency from 400 MHz to 800 MHz. Comparing with Fig.20, as we add more cores to the computation, their “contribution” towards the power consumption far outpace their contribution towards the performance improvement.

Next, we find the relation between total execution time and average power consumption, as shown in Fig.25. The speedup was calculated by dividing the total execution time at 400MHz over the total execution time at 800MHz. The power ratios were calculated by dividing the average power consumption at 800MHz over the average power consumption at 400MHz. Each program has its distinctive speedup behavior whereas the power ratio slowly increases with the number of executing cores. The power ratio is always lower than the
speedup ratio in PI and Cauchy. PI shows that we gain more performance while we incur a much smaller increment on power consumption. Hence, it is always worth operating at a higher frequency for PI. On the other hand, the speedup of matrix multiplication has a higher ratio than the power consumption ratio at the very beginning when we only have small number of cores. Thus, it is worth increasing the number of cores to obtain a higher performance with low level of power consumption. However, the speedup gained by operating at a higher frequency only increases in a very limited fashion, while the power ratio grows much faster than the speedup gained as the number of cores increases. In fact, the matrix multiplication’s power ratio dominates the speedup gained after the number of cores exceeds 20. It is predictable that if we operate at more than 20 cores in this case, it will not be worth increasing the number of cores because of the increment on power consumption. The speedup of the Cauchy program oscillates as the number of cores increases. Overall, it is worth executing at a higher frequency if the speedup gained is greater than the power consumption overhead incurred.

Fig.26 shows the relationship between the number of cores utilized and the overall energy consumption of the entire execution for each workload. We can see that for the PI program the overall energy consumption keeps decreasing as we increase the number of executing cores. This holds true for the embarrassingly parallel program, which is supposed to have the least communication overhead because of its nearly communication free execution.
Also, operating at 800 MHz consumes less energy than operating at 400 MHz for any number of executing cores. Matrix multiplication has a different characteristic as far as overall energy consumption is concerned: the energy consumption decreases up to 16 cores. Beyond this specific point, the overall energy consumption increases constantly as it will consume more energy in order to execute the same amount of work. So 16 cores seem to be the optimal number of executing cores from an energy-aware point of view. Operating at 400 MHz consumes less energy when the number of cores is less than or equal to 24 whereas operating at 800 MHz consumes less energy when the number of cores is greater than 24, even though the difference is small. Therefore, in term of energy optimization, we should run the program at high frequency with a small number of cores and run the program at low frequency with a high number of cores for the matrix multiplication program. Finally, Cauchy does not convince us to increase the number of executing cores at all, since its overall energy consumption never drops below the overall energy consumption of the single core execution case.

Lastly, Fig.27 shows the energy-delay product (EDP) of PI, matmul and Cauchy programs at 800 MHz and 400 MHz, respectively. The energy-delay product is computed from the product of total execution time and overall energy consumption, which illustrates the balance in performance and energy efficiency. Our experiment shows significant differences with the energy-delay product among the programs. First, the energy-delay product of the PI pro-
Figure 27: Energy-delay product comparison of the three workloads at 400MHz and 800MHz.

The program is continuously decreasing as the number of cores increases. It convinces us to add more cores to participate the computation. This is the advantage gained from executing an embarrassingly parallel workload which does not incur additional overhead when we increase the number of cores. Matrix multiplication program has an optimum point at 14 cores on the energy-delay product plot. The energy-delay product rapidly decreases from 2 executing cores up to 14 executing cores. There is some number of cores with a higher energy-delay product within this range. This could be explained by the voltage domain utilization. The program yields a better energy-delay product when there are fewer idle cores. Its energy-delay product continuously increases after 14 cores. When there are more cores participating in the computation, more data transfers have to be done between the root core and other computing cores. The energy-delay product clearly indicates that if we want to maintain the utilization of both performance and power consumption, we should execute the matrix multiplication with 14 cores. We can gain the lowest energy-delay product by executing 8 cores entirely on the first voltage domain and 6 cores on the second voltage domain. Similar to the overall energy consumption in Fig.26, Cauchy does not show any decrease with the energy-delay product. It is not worth increasing the number of executing cores for an embarrassingly sequential workload.
5. Conclusions and Future Work

As many-core processors begin to supersede their single-core counterpart, the methods on how to utilize the full power of these many-core processors have been of emerging interest for programmers at both application and system levels. In this work, we have presented a test drive of the Single-Chip Cloud Computer (SCC), a many-core design prototype by Intel Labs. Three representative workloads with distinctive behavior were analyzed for their scalability on such a research platform. Since SCC employs message passing for inter-core communication, our results indicate the communication overhead among the cores occurred when parallelizing the program could become a bottleneck, which is one of the causes hindering the program scalability on such an architecture. We also investigated the time, power and energy relationship with varying numbers of cores on SCC. Our experiments demonstrate that we lose the edge when increasing the number of cores beyond the point where power consumption overhead incurred is more than the performance speedup achieved.

The study was facilitated by in-depth profiling during program execution. The performance counters measured help indicating the computation and memory access behavior. When the workload has a high computation demand, we should increase the operating frequency to achieve a speedup. Otherwise, we may want to decrease the operating frequency to alleviate the affect from idle clock cycles when the misses and stalls on lower memory accesses are high.

A beneficial observation from the experiment is the appropriate number of cores to be applied to different kinds of workloads. For an embarrassingly parallel workload, our experimental results suggest that we should execute as many cores as possible. For a mixed parallel and sequential workload, there is a specific number of cores that provides an optimum on both performance and energy consumption. It could be assigned to a subset of the cores on the platform, within the same voltage domain if possible. For an embarrassingly sequential workload, it is best to execute on a single core. Our workload analysis approach has demonstrated the usage of available tools and techniques which could be assembled together to study the program behavior on a many-core platform in many aspects.

As far as future work is concerned, we plan to perform fine-grain voltage/frequency assignment to different power domains to study the effect of power and energy consumption on the SCC in more depth. How to perform
auto-tuning of frequency and voltage by monitoring program behavior at runtime is of particular interest. If multiple programs running on a many-core platform, how to schedule them to different power domains based on their performance and energy tradeoff is another avenue we want to explore.

6. Acknowledgements

We would like to thank Intel Labs for providing us with an SCC platform to conduct this research. We also would like to thank Xu Liu from Rice University for providing the HPCToolkit. This work is partly supported by the National Science Foundation under grant numbers ECCS-1301953 and CCF-1065448. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of either Intel or the National Science Foundation.

References


[21] Intel Labs; “SCC Platform Overview,” Intel Many-core Applications Research Community, Revision 0.75, September 2010


[38] Swinburne, R.; “Intel Core i7 - Nehalem Architecture Dive,” www.bit-tech.com, 3 November 2008


