Interactive visualization and dynamic task management of many-core systems
A case study: The Intel Single-chip Cloud Computer

Jimi M. van der Woning
Fibigerstraat 2HS
1097 KA Amsterdam
The Netherlands
tel. +31 (0)6-34440999

Jimi.vanderWoning@student.uva.nl
Student ID 6061400

June 11, 2012

Supervisors: Roy Bakker (CSA, UvA), Andy D. Pimentel (CSA, UvA)
Signed: Andy D. Pimentel (CSA, UvA)
Abstract
As Moore’s Law still applies, we will quickly advance from the current multi-core systems to many-core systems. Some challenges arise for visualizing and managing such a system. There are no available programs that can provide a user with both a performance overview of the entire chip and per task usage statistics, as well as a management system for the many-core system.
For these reasons, a many-core visualization and management tool, dubbed ManyMan, has been developed. Usability tests pointed out that this tool provides an intuitive way to interactively manage a many-core system. ManyMan has been tested with the Intel SCC, a 48-core research chip. The BLCR library has been used for task migration, which turned out to be a time consuming process.
# Contents

1 **Introduction** 7  
   1.1 Outline ........................................ 8  

2 **Related work** 9  
   2.1 scckit ......................................... 9  
   2.2 Momentics ...................................... 9  
   2.3 TuningFork .................................... 10  
   2.4 Condor ....................................... 11  
   2.5 SVP .......................................... 11  

3 **Hardware** 13  
   3.1 Intel SCC .................................... 13  
      3.1.1 Chip overview ............................ 14  
      3.1.2 Memory and network .................... 15  
      3.1.3 Power management ....................... 16  
   3.2 Multi-touch G3 .................................. 16  

4 **Software** 17  
   4.1 BLCR ......................................... 17  
   4.2 SSH Connection Sharing ..................... 18  
   4.3 Kivy ......................................... 18  

5 **Implementation** 19  
   5.1 Back-end .................................... 20  
      5.1.1 Monitoring ............................... 20  
      5.1.2 Task creation ............................ 20  
      5.1.3 Task migration ............................ 21  
      5.1.4 Task pausing / resuming ................. 21  
   5.2 Communication .................................. 22  
      5.2.1 Protocol ................................ 22  
      5.2.2 Security ................................ 23  
   5.3 Front-end .................................... 23  
      5.3.1 Chip overview ............................ 23  
      5.3.2 Help popup ................................ 25  
      5.3.3 Core view ................................ 25  
      5.3.4 Task view ................................ 27  
      5.3.5 Task creation ............................ 27  

Jimi van der Woning 5
6 Evaluation
6.1 Back-end ........................................... 29
  6.1.1 Running latency ................................ 29
  6.1.2 Checkpoint/Restart latency ..................... 30
  6.1.3 Connection Sharing time gain ................... 32
6.2 Communication ..................................... 32
6.3 Front-end .......................................... 34
7 Conclusion ........................................... 35
8 Future work .......................................... 37
A Using ManyMan ....................................... 43
  A.1 Back-end .......................................... 43
  A.2 Front-end .......................................... 43
B Configuration ......................................... 45
  B.1 Back-end .......................................... 45
  B.2 Front-end .......................................... 46
C Communication protocol .............................. 49
D Usability test ........................................ 51
CHAPTER 1

Introduction

Starting with the first ever built computer, people have always wanted to get more out of one than possible at that time. However, as the processing power of computers increases, so does the need for it. Up to the early 2000s, this need could easily be fulfilled by decreasing the size of transistors, by which more of them could be placed on a single chip. In the second half of the past decade, this changed. With current transistor technology, it is very hard to regulate energy flows when transistors get even smaller than they are today. This causes a lot of power to be wasted, which is a problem in both portability and environmental aspects. When decreasing the size of transistors is no longer possible, one could think about increasing the clock speed of the processor. This however causes transistors to use more energy as well, by which they generate more heat. Unfortunately, thermal-design techniques cannot advance fast enough to keep pace with the fast increase in heat generation [10].

To overcome the problems with single-core chips, chip manufacturers have created multi-core chips. We now see that the number of transistors on a chip still doubles every eighteen months, which means that Moore’s Law still applies [24]. However, as the number of transistors on a chip grows exponentially, so do the cost and time required for the logical design of these chips. This problem can be solved by creating a much simpler chip, consisting of many small, simple cores: a many-core chip.

Intel has accommodated research on such chips in its Tera-Scale project. In 2009, this project produced its second processor, which provides 48 fully functional cores, each of which can be seen as a separate computer. The chip is therefore called the Single-chip Cloud Computer (SCC) [41, 14]. The SCC has been created entirely for doing research on how future many-core processors may work, so providing real performance is not necessary. It could therefore be made out of 48 Pentium 1 (P45C) cores, which were originally created in 1994 [18].

The University of Amsterdam was selected to receive one of the SCC systems. Currently, all of its 48 cores independently run a Linux OS that has been developed especially for the SCC: sccLinux. The chip is connected to a management PC, from which all cores can be controlled and monitored [2].

For monitoring many-core chips like the SCC, a challenge arises. With this many cores, it becomes hard to get an overview of what is happening when and where. One wants to know the total payload of the system, as well as the resource usage of a single
core or process, but no view is available that combines these things. Therefore, a tool has to be developed that can provide any user with all performance information that he or she wants. For optimal interactivity, this tool will be optimized for usage on a multi-touch table, while it can still be controlled using a normal mouse and keyboard.

Besides just monitoring performance, one might also want to control what is happening on each of the cores. All in all, the user of the tool should be able to:

1. Monitor the system:
   - Current load and state of the chip.
   - Current resource usage for each core.
   - Resource usage per monitored process on a core.
   - Basic (text) output per monitored process.
   - Overview of running and waiting tasks.

2. Manage the system:
   - Easily create a task.
   - Start a task on a single specified core, or the best core available (suggested by the system).
   - Migrate tasks to other cores, with or without suggestions from the system.

Both parts have to be combined, so that the user knows the status of the system while managing it.

Also, tasks can generate output. When output cannot be accessed, running a task is usually pointless. Finding the best way to provide output to the user is beyond the scope of this project, but it has to be provided in some way.

In this thesis, the tool that has been created for monitoring and managing the Intel SCC will be described. Its results will be shown and several possible extensions to it will be proposed.

1.1 Outline of this thesis

In Chapter 2, the research that has already been done in this area of Computer Science will be discussed. Chapters 3 and 4 discuss the hard- and software tools that have been used in this project. The tool that has been implemented will be described in Chapter 5 and evaluated in Chapter 6. The conclusion follows in Chapter 7, after which some possible future work will be discussed in Chapter 8.
CHAPTER 2

Related work

Not a lot of research has been done in the area of visualizing and managing many-core systems. In fact, the most relevant Google search result is an open PhD Studentship on Visualizing Multicore Performance at Trinity College Dublin [6]. This indicates the novelty of this subject.

2.1 sccKit

Along with the SCC, Intel provides a set of software tools, called sccKit. sccKit consists of command-line tools and sccGui, where sccGui is the graphical version of the command-line tools. The command-line tools contain sccBoot, by which SCC cores can be booted, sccDump, by which the SCC’s memory can be read, and sccBmc, the tool that can initialize the chip and power it on and off. The most advanced graphical tool is sccDisplay, by which X-forwarding can be accomplished, mouse and keyboard interactions can be forwarded and sounds can be played. As, in this project, a management and visualization tool needs to be developed, sccKit’s most interesting tool is sccPerf. This is a visual performance meter for the SCC [2] (see figure 2.1).

One might think that this performance meter takes away the work of creating a visualization tool for the SCC. However, it does not. There is nothing more than the screenshot shows, which means that there is no information per process or whatsoever. Besides that, the code is written in such way that expanding it would be difficult. Finally, the conclusion has been drawn that sccPerf cannot be used in the many-core monitoring and management tool that will be developed. It does however provide a starting point from which the multi-touch interface can be designed.

2.2 QNX Momentics Tool Suite

QNX Software Systems has developed an IDE for writing software for embedded systems, called the QNX Momentics Tool Suite. Momentics provides profiling tools that give users insight into system behaviour and specific tools for multi-core systems [32].

Inside Momentics, special multi-core visualization tools can be found. These tools analyse how the multi-core system acts as a whole and provide information about hard-
ware interrupts, kernel calls, scheduling events, thread-state changes and various forms of IPC [23]. However, the tools have been developed for designing and debugging software for multi-core systems and are not considered for the management and monitoring of a running system.

2.3 TuningFork

IBM’s TuningFork [1] is an on-line visualization and analysis tool for real-time systems. When provided a trace file, it can show the user a wide range of figures that contain information about the system. These can be, for example, the percentage of used memory or time intervals when the garbage collector is off.

Being able to visualize all these aspects makes TuningFork a great tool for debugging
and improving applications. However, like Momentics (see section 2.2), it cannot be used on a running system, since it relies fully on post-mortem trace files.

2.4 Condor

Condor is a specialized workload management system for compute-intensive jobs [5]. It provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring and resource management. When a user wants to run a task, Condor places it into a queue, chooses when and where to run, monitors its progress, and finally informs the user upon completion.

Condor can be used to manage a cluster of dedicated compute nodes or a regular set of desktop workstations. For the use with desktop workstations, special mechanisms have been designed that enable it to effectively harness wasted CPU power from these otherwise idle machines. For instance, Condor can be configured to only use desktop machines where the keyboard and mouse are idle. When Condor detects that a machine is no longer available (e.g. a key press or mouse move detected), it is able to transparently produce a checkpoint of all the machine’s running jobs and migrate them to different machines.

Having a shared filesystem across Condor’s nodes is not required. When no such filesystem is available, Condor can transfer a job’s data files on behalf of the user, or it may even be able to transparently redirect all the job’s I/O requests back to the machine the job originally ran at. As a result, Condor can be used to seamlessly combine all of an organization’s computational power into one resource [4].

Even though Condor provides all scheduling and relocation functionality that is needed for a many-core visualization and management tool, it cannot simply be used. First of all, the level of resource monitoring provided by Condor is not extensive enough for the system. It does provide a total overview of the cluster’s payload, but cannot provide per workstation or per task performance information.

A more important reason for not using Condor is the fact that, currently, no implementation of it exists for the Intel SCC. As the SCC’s cores run a very basic Linux OS, it is possible that porting Condor to the SCC requires quite some changes, which would probably be a project in itself. Finally, Condor might be too heavyweight for the SCC’s lightweight cores.

2.5 Self-adaptive Virtual Processor

Several mechanisms and frameworks are available to manage multi- and many-core systems on different levels of granularity. One of these is the Self-adaptive Virtual Processor (SVP) [19]. SVP is a concurrency model developed at the University of Amsterdam. It provides a method to achieve concurrency, without having to manage it. Using SVP, a group of tasks (i.e. a family of threads) can be delegated to a place, a resource where a task can execute. Such resource could be a core, a group of cores or a complete machine [2].

A working implementation of SVP is already available for the SCC [3]. However, it does not provide performance information, nor does it allow a user to dynamically
relocate tasks. Given this fact, one can conclude that no system has been found that combines visualization and management of many-core systems on the high level that is desired for this tool.
CHAPTER 3

Hardware

In this project, two pieces of hardware have been used. First, there is the Intel SCC, the 48-core chip on which a case study has been performed. This chip will be discussed in section 3.1. Second, the multi-touch table on which the visualization and management tool will run, will be described in section 3.2.

3.1 Intel Single-chip Cloud Computer

Back in 1996, the ASCI Red supercomputer was the first machine to achieve a TeraFlops (one trillion floating point operations per second). The machine took more than 185 square meters in space and consumed over 500 kW of power [41]. In 2006, Intel started the Tera-Scale project. This project focuses on exploring energy-efficient design and core-to-core communication solutions for future many-core chips [13].

The TeraFlops Research Chip (TRC) was the first chip produced by the Tera-Scale project. It was created in 2007, only eleven years after the ASCI Red, and provided the same performance. This chip, however, is no bigger than the average multicore chip that is used nowadays and consumes only 62 Watts. A downfall of this chip is the fact that it consists of only two floating point units. Therefore, it cannot be used as a general purpose device [15, 17].

As stated in Chapter 1, the Single-chip Cloud Computer (SCC) was the second chip produced by Intel’s Tera-Scale project. It was developed in 2009 and made available for research in 2010. The chip provides 48 fully functional cores, each of which can be seen as a separate computer. Hence, the name Single-chip Cloud Computer was chosen [14].

Following the TRC and the SCC, Intel developed a new product, called the Many Integrated Core (MIC) architecture, in 2011 [16]. Its design is based on the TRC, the SCC and the Larrabee many-core visual computing project [34]. The design of this Larrabee chip is based on that of modern GPUs. For MIC’s design, newer Xeon cores have been used. Using this chip, developers can create platforms running at trillions of calculations per second, without having to change a thing in their existing code.
3.1.1 Chip overview

The SCC is a single chip, consisting of 48 Intel IA-32 P54C cores connected by an on-chip mesh network. Intel has decided to use this relatively old type of cores due to the fact that they can operate on low voltages (only 3.3 volts) and they are cheap in silicon design. Besides that, these cores are very small in silicon area, which made it possible to put 48 of them on a single 45nm CMOS chip [2]. The chip has been designed in such a way that, in the future, potentially hundreds or even thousands of cores could be placed on a single chip using this design [40].

Figure 3.1 shows the actual SCC hardware that the University of Amsterdam received from Intel. Currently, it cannot be used as a standalone computer, due to the fact that no SATA driver is present yet. Instead, a management console PC (MCPC) has to be used, which in this case is an HP Proliant ML110 G6 [2]. The MCPC is connected to the SCC using an external PCI Express cable. In order to access one of the SCC’s cores, one has to open an SSH connection to the core in question from the MCPC.

Internally, the SCC is controlled by a Board Management Controller (BMC), which can be used to initialize the on-board FPGA and several communication channels. It can also be used to read the current board status of the SCC, including its power usage. The FPGA acts as the chipset of the system and controls the SATA and Ethernet ports. Besides that, it also acts as a bridge between the internal format and the PCIe interface. By placing an FPGA on the board, the possibility arises to reprogram parts of the SCC’s hardware, without having to physically change it. However, as the storage space of the FPGA is limited, adding a new feature means removing another [2].
3.1.2 Memory and network

In figure 3.2, the network layout of the SCC is shown. It shows that the SCC contains a 6x4 mesh network of 24 tiles in total. Each tile consists of two cores, each with their own cache, and a router for memory access and communication between cores. Besides that, each tile contains 16kB of storage, which is used as a Message Passing Buffer (MPB). The MPB is most useful when sending short messages between cores, as the storage space is limited. All MPBs on the chip can be accessed by any of its 48 cores. However, conventions state that each core may usually only access half of the MPB (8kB) on its own tile, unless a different protocol is used [40].

Every core has 16kB of L1 cache, in which anything from the complete address space, including the MPBs, can be cached. This introduces a possible problem, since the information on the MPBs can be changed by any core. To overcome these problems, a special memory type for MPB data (MPBT) was added to the virtual memory system, together with an instruction to invalidate all of this data. Besides the MPB caching problem, the P54C core originally supports just a single outstanding write request. By adding a Write Combine Buffer (WCB) that combines any adjacent writes to a single one, this issue is solved [40, 2].

The SCC chip is equipped with four DDR3 Memory Controllers (MCs) (see figure 3.2), which each use 34 bit addresses. This allows the system to address a maximum of 64GB, although the university’s SCC system only has 32GB RAM. Using 34 bit addresses is not something trivial for the P54C core, which has only 32 bits of physical address space. To solve this problem, special lookup tables (LUTs) are used. These LUTs can translate the first eight bits of the core’s 32 bit address to the first ten bits of the memory’s 34 bit address [40, 2].
3.1.3 Power management

What figure 3.2 does not show is the fact that the SCC consists of six voltage islands, each consisting of four tiles. Using a Voltage Regulator Controller (VRC), the voltage of each of those tiles can be adjusted from 0V to 1.3V, in steps of 6.25mV [2, 14].

Besides the voltage, the frequency at which the cores of a tile run, as well as that of the routers and memory controllers, can also be regulated. The tile’s frequency can be set in 15 non-linear steps from 100MHz up to 800MHz. The frequency of the routers can be either 800MHz or 1600MHz, where the MCs’ can be either 800MHz or 1066MHz. Intel has measured the chip’s power consumption to be 25W when the cores are running on 0.7V at 125MHz and 125W on 1.14V at 1GHz. Note that running the SCC at 1GHz was only possible in this test. Normally, the SCC can only reach 800MHz [2].

In the current setup, the cores are clocked to 533MHz and the routers and MCs are clocked to 800MHz. This is the system’s default configuration.

3.2 PQ Labs’ Multi-touch G3 Basic

PQ Labs is one of the worldwide market leaders in producing multi-touch solutions [27]. In 2007, they produced their third multi-touch overlay, called the G3 (see figure 3.3). This overlay can be bought in various sizes, which makes it fit on a regular screen of any common size between 32 and 65 inches [29]. At the University of Amsterdam, the screen that has been used for this project was a 42 inch Philips BDT4251VM/32 [26]. It is equipped with the G3 Basic, which is controlled by a Mac mini that runs Windows 7, Mac OS X and Ubuntu Linux.

To recognize a user’s touches, the G3 uses PQ Labs’ own LED Cell Imaging technology, which uses infrared to recognize the touch points. Using this technology, the G3 Basic can recognize up to six simultaneous touches of objects which are at least 6mm in diameter. These objects can be recognized with an accuracy of around 1.5mm and have a response time of only 7 to 12ms [30]. The system comes with drivers for Windows (2000 or newer) and Mac OS X; Linux drivers can be downloaded from the company’s website [31].

Figure 3.3: PQ Labs’ Multi-touch G3. From [28].
In order to create the many-core visualization and management tool, three pieces of software had to be used. Berkeley Lab Checkpoint/Restart (see section 4.1) is used to be able to migrate tasks between cores. To speed up core access, the SSH Connection Sharing mechanism is used (see section 4.2). Finally, the tool has been developed using the Kivy framework, which is discussed in section 4.3.

4.1 Berkeley Lab Checkpoint/Restart

Berkeley Lab Checkpoint/Restart (BLCR) [7] is part of the Scalable Systems Software Suite [33]. It has been developed by The Future Technologies Group [9]. Using BLCR, tasks can easily be stopped (checkpointed) and restarted later. Restarting a task can also be done on other cores or even other machines, as long as these machines have largely the same configuration as the machine the program originally ran on. BLCR has been designed to provide a system-level checkpoint/restart implementation for Linux clusters [12]. Currently, BLCR supports checkpointing both single and multi-threaded applications, as well as process trees, process groups and POSIX sessions. However, it does not provide support for applications that use sockets or SystemV IPC mechanisms, such as shared memory and semaphores [8].

After loading BLCR’s kernel modules, making an application checkpointable is as easy as executing it with the `cr_run` command: `cr_run <executable> [args]`. However, when the application uses statically linked libraries, this is not possible. In that case, BLCR’s library code has to be passed when compiling [8].

When a task is running, checkpointing it can be done using the `cr_checkpoint` command. BLCR will then create a complete memory dump of the process (or process tree) whose Process ID (PID) has been provided, a so called context file. Note that writing this file to disk can be a very time-consuming task. A task may continue to run when checkpointed, but will in this implementation always be killed. After checkpointing and killing, the task can be restarted using `cr_restart <context_file>`. In order to restart, the original executable, as well as all shared libraries and used files need to be present at the exact same location. Besides that, the PIDs of the checkpointed processes need to be free, or otherwise the `--no-restore-pid` flag has to be used to obtain new PIDs [8].
BLCR runs on Linux kernel versions 2.4.x and 2.6.x [12]. This means that, in order to be able to run BLCR, the SCC had to be downgraded from kernel version 3.0.x to version 2.6.x. Currently, the SCC has BLCR’s latest version (0.8.4) [7] installed.

4.2 SSH Connection Sharing

When opening multiple SSH connections to the same machine, which is needed to start multiple tasks on a single SCC core, the SSH Connection Sharing mechanism can be used. By using this mechanism, any connection that is made after the first will reuse that first connection [36]. This way, those subsequent connections can be established much faster, which is partially due to the fact that one has to authenticate only once. However, it also means that the master (i.e. first) connection needs to remain active until all child connections are closed.

When using Connection Sharing, all of the connections share the same TCP connection. This means that all of them use the same SSH options as the master connection, which might create some problems [36]. For example, when X-forwarding is not enabled on the master connection, none of the child connections will be able to enable it either. Also, Connection Sharing could introduce minor problems with programs that require a lot of data transfer over the connection, such as scp.

4.3 Kivy

Kivy [21] is an open source Python framework for rapid development of applications that make use of innovative user interfaces, such as multi-touch applications. It is part of the Natural User Interface [25] group, a global research community focused on the open discovery of natural user interfaces [35].

Kivy is fast in both application development and application execution speeds. The high abstraction level allows users to easily write programs, while time-critical functionality has been written on the C level to leverage the power of existing compilers. *Intelligent algorithms* try to minimize the cost of intensive operations and the GPU is used when possible [22]. Kivy’s graphics engine has been built with OpenGL ES 2.0, using the modern and fast way of doing graphics [21]. This way, lots of actions can be performed at the same time. For example, it is possible to tap a button, scroll a list and zoom in on a photo at the same moment. This makes Kivy not only a multi-touch, but also a multi-user system [38].

A large variety of devices can be used to run Kivy. The same Kivy source code runs not only on Linux, Windows and Mac OS X, but also on both Android and IOS [21]. This flexibility allows Kivy to easily adapt to new technologies or touch drivers. Currently, Kivy supports Windows’ WM_TOUCH drivers, drivers for Apple’s Multi-Touch capable devices and HID kernel input events for Linux support. Besides these default drivers, Kivy also supports TUIO [37] and a number of other input sources [22].

Kivy is a community project, led by professional software developers. A small group of five core developers is responsible for developing and supporting Kivy, alongside the community. Some of the developers also work for companies that use Kivy for their professional products [20]. This is possible since Kivy is completely free to use, under a LGPL 3 licence [21].
The many-core visualization and management tool has been dubbed ManyMan, as in Many-core Manager. ManyMan consists of two main parts, a front- and a back-end, with a communication layer in-between (see figure 5.1). The reason for this separation of front- and back-end is twofold. First, the front-end could theoretically be attached to a different many-core chip, or, the other way around, a different front-end could be attached to this back-end. This increases the usability of the tool, since it is not restricted to just one chip or interface. The second reason for this separation is of a more practical kind. At the University of Amsterdam, the SCC is located in a server room where no monitor could be easily attached to it. With the separated front-end, one does not have to worry about where to put the monitor, which can now be located practically anywhere (see section 5.2.2).

Instructions on how to setup, start and use ManyMan can be found in appendix A.
5.1 Back-end

The back-end has been written completely in Python, for which there are two main reasons. First, Python is a relatively easy programming language which allows for rapid development. It also forces users to write clean code, since blocks of code are indicated by indentation. Second, Python provides good support for running many threads, starting shells on remote machines using subprocesses and TCP communication. One might say that Python does not deliver the best performance, but that is not an issue for this proof of concept.

5.1.1 Monitoring

Monitoring the chip is half of the visualization and management tool. One would like to know the status and payload of every core and task on the system. Unfortunately, the SCC does not provide such information about the chip as a whole, which means that it needs to be retrieved separately from each core.

In order to access one of the SCC’s cores, one needs to open an SSH connection to the core in question from the SCC’s MCPC (see section 3.1.1). Since an SSH connection needs to be opened for each task that runs on the SCC, SSH Connection Sharing (see section 4.2) is used. The monitoring process runs throughout the entire program, which makes it the perfect candidate to be the master connection.

In order to obtain the payload information of each core, the Unix `top` command is used. At adjustable intervals, `top` provides information about all processes that are running on the core and the total payload of the core itself. The fact that this information is everything that has to be shown, makes `top` the ideal monitoring solution. However, it has to be noted that `top` might create some overhead, since it accesses more information than just that information that is needed. For example, information about kernel processes is retrieved as well, which is in this case useless. It would therefore be better if the process information would be read directly from the contents of the `/proc` folder, but that would require a lot of additional work. Then again, performance is not the main issue in this proof of concept.

It has to be noted that any resource usage of processes that have not been started using ManyMan, as for example kernel processes, will be marked as overhead. This overhead will be visible in the total core payload, but, of course, not in the per task payload. Because of that, the task payloads will not add up to the total core payload. The overhead tasks can unfortunately not simply be added to ManyMan’s task list, as these tasks can possibly not be stopped and almost certainly not be checkpointed.

5.1.2 Task creation

When a task needs to be created, a child connection is added to the monitoring’s master SSH connection (see section 5.1.1) of the core on which the user has decided the task should run. On this child connection, the task is started with BLCR’s `cr_run` command (see section 4.1). When the program starts to run, its output will be buffered. It will be sent to the front-end upon request.

In order to be able to checkpoint the task, its Process ID (PID) has to be known. Getting this is not something trivial, since one is only able to retrieve the PID of the SSH
command. This PID can easily be found by executing `echo $$` on the SSH connection. Since the started task is a child process of the SSH command, one can retrieve its PID by looking for the SSH connection’s child processes. This can be done using Unix’ `ps` command with the `-a -o pid,ppid,comm` flags, which will list all running processes with their PID and their parent’s PID (PPID). The `cr_run` command will be the last one executed on the SSH connection, which means that its PID will be the latest of those with the SSH connection’s PID as their PPID. This way, a task’s PID. can be retrieved.

In case a user does not know which core to start a task on, a `smart-start` function has been implemented. When smart-starting a task, the core with the least CPU and memory usage is selected. In this process, both the CPU and the memory usage have the same weight. A possible growth in CPU or memory usage is foreseen by also taking the number of running tasks on a core into account. The more tasks are running on a core, the smaller the chance a task will start there gets. As soon as the best core to run the task on is found, the task will be started on that core as usual. Note that this smart-start function does not keep track of the history of core usage or whatsoever, but it just looks at the current core state. This means that there is some room for improvement here.

### 5.1.3 Task migration

A task that needs to be migrated will first be checkpointed using the `cr_checkpoint` command from the BLCR library (see section 4.1). The location of the context file that is hereby created will be stored in order to be able to restart the task later. When checkpointing is complete, the task can be restarted on the desired core using `cr_restart`. When a user does not want to restart the task yet, it will instead be moved to a list of stopped tasks. Since all cores of the SCC mount the same `/shared` directory, one does not have to worry about sending context files among cores. These can just be found on the exact same path as where they were originally stored. One does however need to worry about the task’s PID. When simply executing `cr_restart`, BLCR will try to restore the task’s original PID. When this is unavailable, the task will not be able to restart. In order to prevent accidents from happening, the `--no-restore-pid` flag (see section 4.1) is used. Now, the program will be assigned a new PID when restarting.

Just like the smart-start function described in section 5.1.2, a `smart-move` function has been implemented. Using this function, a task can be moved to the best possible core, which is found in practically the same way as it is done when smart-starting. The only difference here is the fact that the best core will never be the core the task is already running on. The decision for leaving out this core as an option has been made because the user wants to move the task away from the current core. Even when the task’s current core is the least busy of all cores, the task will always change cores.

### 5.1.4 Task pausing / resuming

Since checkpointing a task produces a lot of overhead, tasks can also be paused using the traditional POSIX STOP signal, after which they can be resumed by sending the POSIX CONT signal. It has to be noted that even though the process is paused and will not use the CPU, it will not release resources such as memory. Due to this fact, paused tasks cannot be moved to other cores, as long as they are not checkpointed. Besides that, not releasing memory might be a problem for the SCC cores, since their private memory is
limited (around 640MB). For manually scheduling CPU intensive tasks however, this is a great solution.

5.2 Communication

The communication layer makes use of TCP connections. This way, one does not have to worry about whether messages arrive or not, since TCP takes care of that. Also, Python provides good and easy TCP support, which combines easily with the front- and back-end. A downfall of TCP is its overhead, which makes it less useful to send large amounts of real-time information. This, however, is not really an issue here, since messages shall usually never be sent more often than once a second and their sizes rarely exceed a few kilobytes. An exception has to be made here for task output messages, which may be a lot larger in size. In order to prevent these messages from clogging up the system, there is a (configurable) maximum number of output lines that may be sent at once. By default, this number is set to 100.

Across the TCP connection, messages are sent in the JSON format. This format is human-readable, which allows for easy debugging. Besides that, machines can also quickly encode and decode JSON messages and, as JSON is a communication standard, it allows for great portability. The extra data that JSON messages carry around might create some overhead, but still, performance is not the main issue in this proof of concept.

5.2.1 Protocol

Communication between the front- and back-end follows a certain protocol, of which the main part is shown in figure 5.2. At first, the front-end will initialize itself by sending a client_init message containing information about itself. The back-end will then respond with a server_init message that contains information about the SCC, such as its number of cores. Since knowing this information about the cores is vital for the system to run, no other messages may be sent before initialization is complete.

After initialization, the status of the entire SCC system, i.e. per core and per task performance, will be sent to the front-end in a status message. This process will be repeated every second, as the performance information is updated once a second as well.
Contrary to the status message, which will be sent nonetheless, a task’s output will only be sent upon request. This task_output_request message had to be introduced, because otherwise sending all output would clog up the connection. Besides, a user will only be able to see the output of a limited number of tasks at the same time.

The most useful messages are those prefixed with task_. As expected, these messages are sent from the front-end when a task has to be started, stopped, moved, etcetera. These messages are not replied to by the back-end, since the changes they trigger will already be visible in the next status messages.

A complete overview of the entire message protocol can be found in appendix C. There, the exact contents of all message types are described as well.

5.2.2 Security

At the start of this project, the SCC could only be accessed from the internal cabled network of the Science Park facility of the University of Amsterdam. This was a problem, since the multi-touch table is not connected to that network, but to the university’s wireless network. Therefore, during this project, a single port has been opened on the SCC’s MCPC to give the multi-touch table the ability to communicate with it.

As this network can only be accessed by students and employees of the university, no security had to be implemented. It should however be noted that this creates a possible threat. Any student or employee that knows the communication protocol can now potentially execute tasks on one of the SCC’s cores with root permission, as long as ManyMan’s back-end is running. Therefore, when this software will be used for more delicate systems, some encryption and authentication will have to be implemented.

5.3 Front-end

The front-end of ManyMan has been developed using the Kivy framework (see section 4.3). As listed there, Kivy has been written in Python, which makes it easy to choose Python as the programming language for the rest of the front-end as well.

5.3.1 Chip overview

The chip overview as a whole is shown in figure 5.3. On the left side, the cores are presented in the order they are physically arranged at on the chip. This order serves no functional purpose, but has just been chosen to provide the user a realistic view of the chip. The payload of each core is visualized by a coloured overlay that changes in both colour and size (see figure 5.4a). The portion of the core that is taken by the overlay literally translates to the core’s CPU usage, where the colour changes from green at 0% CPU to red at 100% CPU. Although the information is only updated once a second, the overlay is animated to fade to the new payload within that second. This greatly improves the looks of the interface.

In one of the usability tests (see section 6.3), the test subject suggested that it would be nice if the number of active tasks was shown per core in the chip overview. Below the name of the core, this information can now be found. More detailed information about
a core’s payload can be shown by tapping the core in the interface. A popup will open, of which the contents are described in section 5.3.3.

On the right side of the window, a list of tasks is shown (see figure 5.4b). These tasks are currently not running on any core, but are either not started yet or have been stopped by the user. In order to (re)start such a task, one can simply drag it to the core he or she wants it to run at. If the user does not care on which core the task will run, the task may be smart-started (see section 5.1.2) by tapping the play button on the right side of the task. By clicking the copy button on the left side of a task, the task can
be duplicated. This is especially useful when said task is a benchmark program, since running multiple instances of a task at the same time usually serves no purpose.

5.3.2 Help popup

During the usability tests (see section 6.3), some test subjects proposed the idea of adding an instruction popup to the system. This would be especially useful for explaining the icons, but also instructions on how to start and move tasks were thought to be useful. ManyMan’s help window is shown in figure 5.5. It can be opened by pressing the Help button on the right side of the chip overview. In this popup, one can also find information about the creator of the tool and the institute on which the tool has been developed (in the About tab), as well as licensing details (in the License tab).

5.3.3 Core view

When tapping a core in the chip overview, a popup like the one in figure 5.6 will open. On the left side, the history of the CPU and memory usage is shown. In these graphs, the white line indicates the total load of the core, which consists of all tasks started by ManyMan, plus all overhead. The coloured lines in the performance graphs indicate the payload of the tasks that have been started using the many-core management system. The colours of these lines match the colours of the tasks in the task list on the right side of the core view.

The tasks in the task list can be moved to a different core by simply dragging them to the core a user wants them to run on. When dragging a task, all open core popups will swerve out of the way so that they do not block any core. The smart-move (see section 5.1.3) option, along with some additional controls and information, is located in the detailed task view, which is described in section 5.3.4. This task view can be opened.
by tapping the information button on the left side of a task. When a task is dragged to anything that is not a core, as for example the task list on the side of the main view, it will be checkpointed and moved to the chip overview’s task list (see section 5.3.1).

In order to be able to compare the payloads of two or more cores, multiple core views can be opened at once. The user may drag them around to prevent them from lying on top of each other. The popup can also be scaled to fit more of them on the screen, or rotated for when the user wants to look at it from a different angle.

Figure 5.6: ManyMan’s core view, containing information about core 14.

Figure 5.7: ManyMan’s task view, on a finished task ‘Memory’.
5.3.4 Task view

The detailed task view (see figure 5.7) looks similar to the core view discussed in section 5.3.3. Again, the left side of the popup contains information about the CPU and memory usage of the task. On the lower right side of the window, the last 100 lines of the task’s output are shown. This number is configurable, but cannot be too large due to Kivy’s inefficient way of rendering text. The complete output of a task is written to a file, so that it can be accessed later.

Above the output, the task control buttons are shown. Tapping the stop button will signal the back-end that the task needs to be checkpointed, after which the task will be moved to the chip overview’s task list (see section 5.3.1). When a user wants to temporarily pause a task, he can tap the pause button. The back-end will then send a POSIX STOP signal (see section 5.1.4) to the task, after which the pause button will be replaced by a resume button. Tapping this button will cause the task to be resumed by sending the POSIX CONT signal to it. Finally, tapping the move button will \textit{smart-move} the task to the best available core other than itself (see section 5.1.3).

Just like the core views (see section 5.3.3), multiple task views may be opened at once to compare their performance. It is even possible to have both multiple core and task views open at the same time, which is useful when both tasks and cores need to be intensively monitored.

5.3.5 Task creation

When tapping the \textbf{Add task} button in the main window (see figure 5.3), a popup will open in which a command and optionally a name can be entered. This popup is shown in figure 5.8. For multi-touch support, an on-screen keyboard can be used to enter the

![Figure 5.8: The task create popup along with the on-screen keyboard.](image)

"Jimi van der Woning"
name and command of the task. When no name is entered, the name of the binary that is executed will be used as the task’s name. After the create button is pressed, the task will be added to the task list of the main view (see section 5.3.1).

A better way of creating a task would be by selecting a binary using a file browser. Unfortunately, the front-end does not run on the SCC’s MCPC, which means that one cannot just open a graphical file browser and navigate to the file. Even though remote file browsers such as sshfs are available, these cannot be used within Kivy. Manually implementing a remote file browser seems the best solution, but this would require quite some additional work. Unfortunately, implementing such file browser does not fit within the scope of this project. Because of that, no file browser has been implemented. Besides, manually entering the path to the binary works fine for now.
The evaluation of ManyMan consists of two parts. For testing the back-end and communication parts, several experiments and benchmarks have been performed, which are discussed in sections 6.1 and 6.2. The front-end has been tested by performing some usability tests. The results of these tests can be found in section 6.3.

6.1 Back-end

ManyMan’s back-end has to perform three main tasks: chip monitoring, task creation and task migration. For monitoring, the default Unix top command is used (see section 5.1.1). This program simply does its job, so there is nothing to test on the monitoring section. For the task creation process, it is interesting to know how much overhead starting a task using cr_run (see section 5.1.2) will generate, if any. The results of tests that have been performed on this subject can be found in section 6.1.1.

The most interesting back-end process is that of task migration (see section 5.1.3). The cr_checkpoint and cr_restart commands that are used here are tested and working, but both tend to require some time to execute. Tests have been performed on how much time this actually is. These tests are described in section 6.1.2.

In order to perform its tasks, ManyMan makes use of SSH Connection Sharing (see section 4.2). Tests on how high the time gain really is are presented in section 6.1.3.

6.1.1 Running latency

In order to be able to stop a task in mid-execution, tasks are started using the cr_run command (see section 5.1.2). Since these tasks will be constantly monitored, executing one this way might generate some overhead. In order to test this, experiments have been performed using a program that calculates the sum of some million random numbers. The Unix time function has been used to time this program when it is executed both with and without the cr_run command. On the SCC, the time function is unreliable for measuring the wall-clock time, due to the varying frequencies of the cores. In a period that the frequency does not change, however, this function can be used to measure relative times.
In figure 6.1, the average time it takes to add a hundred million random numbers on SCC core 2 is shown. It shows that running the program using \texttt{cr\_run} introduces a minor overhead of just about 0.2%. Even for the larger tasks, this difference will probably not even be noticeable.

### 6.1.2 Checkpoint/Restart latency

As described in section 4.1, checkpointing can create a lot of overhead, as the BLCR library writes the complete state of the process to the filesystem. On the SCC, the only (persistent) filesystem is NFS mounted from the MCPC. This allows easy migration of tasks across cores, but also introduces a large latency for task migration.
Figure 6.2 shows the latency for checkpointing and restarting processes with increasing memory usage. The testing program that has been used here simply allocates a specified number of megabytes of memory. This memory is then filled with random data, after which the sum of this data is calculated. While calculating the sum, the process will be checkpointed. It is made sure that all requested memory has been allocated and filled with random data at that time. This way, problems with lazy allocation of memory pages will not be encountered.

The measurements in this experiment were done by adding a timing mechanism in the BLCR source code. At the beginning and end of the main function, the time stamp counter (TSC) is read and the values are subtracted. To convert to seconds, the resulting value is divided by the core frequency. As expected, the time required for checkpointing scales linearly with the amount of allocated memory. The actual time for checkpointing highly depends on the application and its current state.

Besides the test where context files were written to NFS on the MCPC, an additional test has been performed where the context files were written to the filesystem in RAM (/tmp). The results of this experiment can be found in figure 6.3. It shows that the time required for checkpointing can be reduced with approximately 4.5%. When restarting a task from RAM, the difference is much bigger. A task now restarts almost twice as fast, with a speedup of 49.3%. This giant difference is due to the fact that loading data from NFS into RAM is a very time consuming task. Memory access could be improved by using different page table flags as described in [40], but this would require hacking into the SCC kernel.

When context files are written in RAM, there is a problem when tasks need to be migrated. As multiple cores do not share their RAM, the context files would have to be copied between cores. This could possibly be done by using Copy Cores or shared memory (memory remapping). However, relocating context files would slow down and complicate the restarting process again.
6.1.3 Connection Sharing time gain

In order to speed up each access to a core, SSH Connection Sharing is used (see section 4.2). Figure 6.4 shows the average time it takes to open an SSH connection to SCC core 10, both with and without using Connection Sharing. In order to obtain these results, an `ssh` command was used to open a connection to core 10, on which immediately the `exit` command was executed. In the same way as the latency of `cr_run` was measured (see section 6.1.1), the Unix `time` command has been used. In this case, however, the results of this command are accurate, since the experiments were performed on the SCC’s MCPC, which does have a valid `time` function.

From figure 6.4, one can see that there is an average speedup of 0.61 seconds when using Connection Sharing. Half a second might not seem a lot, but when keeping in mind this speedup is achieved each time a task is started, one can conclude that it will easily grow into seconds or even minutes of time gain.

In order to make sure these results were not distorted due to immediately closing the connection using the `exit` command, additional experiments have been performed in which a `sleep` of 10 and 20 seconds has been executed. For these experiments, the average time gains were 0.62 and 0.61 seconds respectively, by which the initial measurement is confirmed.

6.2 Communication

In order to get an idea of how much data can be sent between front- and back-end, the communication latency and throughput have been measured. To obtain the communication latency, the time it takes to send a short message (PING) over a TCP connection from front- to back-end and back is measured. The results of this test are shown in figure 6.5. It is shown that the average time to ping is about 18.0 milliseconds, with a standard deviation of 23.6 milliseconds. This means that in around two third of the cases, pinging takes less than 42 milliseconds and that it takes less than 65 milliseconds 95% of the time. From the figure, a few higher results at around 170 milliseconds can also be seen. This is still a somewhat acceptable result, but in two cases, the latency rose up to almost three seconds. Unfortunately, these results are due to the way the
university network has been designed and protected. It is however impossible to pinpoint the exact cause of the peaks, as network designs have become highly complex.

For measuring the throughput of the TCP connection between front- and back-end, messages of 10 MB (consisting of only a’s) have been sent from the SCC’s MCPC to the multi touch table. This test showed that the average communication speed lies around 333.7 kilobytes per second, with a standard deviation of 59.3 kB/s (see also figure 6.6). This is quite a bit lower than one would expect from a network on which Linux distributions can be easily downloaded at around 3 megabytes per second. Again, pinpointing the exact cause here is impossible. It could possibly be the case that, in order to communicate with the MCPC over the wireless network, messages have to go through so many layers of security that they simply cannot arrive any faster. However, it is also possible that, during all tests, someone else was using up the bandwidth of the access point, even though the tests have been repeated multiple times.

Figure 6.5: Time to ping from the multi-touch table to the SCC’s MCPC for 100 measurements.

Figure 6.6: Communication speed when sending 10MB of data from the multi-touch table to the SCC’s MCPC for 100 measurements.
6.3 Front-end

In order to test the usability of the front-end, a few Computer Science students and a couple of students from non-computer related disciplines were asked to perform a number of tasks. After these tasks had been completed, the students were asked a number of questions about the usability of the software. The tasks that had to be performed and the questions that have been asked can be found in appendix D.

The general opinion of both the Computer Science and the non-Computer Science students was that the tool looked great. They all found the way tasks have to be started very intuitive and really liked the detailed core overview. Most of them thought the total chip overview was very clear as well, although one Computer Science student rightfully noted that no information is provided about the memory usage of the system as a whole. He also noted there was no way to see how many tasks were running in the chip overview, as this was not present at the time the tests were performed. Thanks to his suggestion, each core image now contains information about the number of running tasks.

The most difficult step for the Computer Science students was the one where a task had to be duplicated. One of them had mistaken the smart-start button for the duplication button, where the rest tried to copy a task by double-tapping or a down pulling gesture. Replacing the icon with something like \( 2x \) was suggested by a student, but, when asked, the other students thought the current icon was better. They preferred adding a Help menu in which both the entire system as the icons were explained. Notably, the non-Computer Science students immediately identified the icon as the one that had to be used for task duplication.

Task movement was the hardest task for the non-Computer Scientists. When moving a task, one has to press it for half a second before he or she can start dragging it. This is caused the fact that Kivy does not handle the touch events any sooner, due to the design of this multi-touch framework. The Computer Science students understood this without any need for explanation, but it was hard to explain it to the rest. Unfortunately, as this problem is part of Kivy’s implementation, changing this would require modifying the framework, which would be a lot of work.

Finding the detailed core-view did not create any problems for anyone, neither did finding the task output or controls. Some people thought that although the status of a task changed, pressing the control buttons did not really show a response. Most, however, thought that creating a better indication, like, for instance, a popup confirming the button press, would only be annoying. They thought that showing the status update in the title of the detailed task-view was enough.

A thing that confused some people was the order in which the cores are shown. As discussed in section 5.3.1, the cores are presented in the order in which they are physically present on the chip. As this order is not just 0-47 from left to right, top to bottom, most test subjects struggled in finding specific cores.

Creating a task did not create many problems either. One of the test subjects even liked the keyboard, as it worked better than the one on her mobile phone. Another missed an OK button, although, when explained, she understood that the Enter key had to be used for this purpose.
Before this project had started, no program existed that provided both interactive visualization and dynamic task management for many-core chips. The best visualization tool around was Intel’s scPerf, but that did not provide any functionality to manage the chip. Other visualization tools are mainly focused on visualizing post-mortem trace files and are not considered for a running system.

A good management system that is currently around for many-core systems is SVP. This software, however, does not provide visualization tools, nor does it allow for easy task migration. Condor is a framework that does provide task migration possibilities, but this system will have to be ported to the SCC. For this reason, the many-core management tool ManyMan has been developed. A case study has been performed on using this tool on the Intel SCC, a 48-core concept vehicle that has been developed for doing research on future many-core chips.

For task migration between the SCC’s cores, the BLCR library has been used. This library provides an easy way to stop tasks and restart them later, possibly on a different core. Tests performed on the latency of these operations showed that the time needed for those linearly increases with the amount of memory that a task uses. As the combination of checkpointing and restarting a task that uses 256 MB of memory takes over a thousand seconds, moving such tasks should only be done when one is absolutely sure about the need to move the task. However, for tasks that do not use a lot of memory, relocating does not take that much time and can be done more often.

By using the SSH Connection Sharing mechanism, the process of connecting to an SCC core has been sped-up. Where connecting usually takes over half a second, connecting using Connection Sharing takes just a tenth of a second. Since a new connection has to be opened to an SCC core each time a task is started, Connection Sharing quickly shows its time gain.

The university’s internal network and its security turned out to be the major bottleneck of the system. The communication latency was not that bad, although the outliers of about three seconds might cause the performance visualization part to not be able to visualize everything. More importantly, the limited throughput of just 333 kB/s might clog up the system when tasks start to generate lots of output. This has been prevented by limiting the amount of output lines that can be sent at the same time and only sending the output of tasks of which the detailed view is opened, but this is not the preferred
solution. Unfortunately, changing the networks security policy is not something that can just be done, meaning that these shortcomings cannot be improved with the current setup.

Usability tests have pointed out that ManyMan is an intuitive system for visualizing and managing a many-core chip. Both Computer Science and non-Computer Science students were able to properly deal with the tool. Most suggested improvements for the system have been implemented, making the tool to what it is today. Finally, all test subjects noted that the tool looked great and provided them an intuitive way of monitoring and managing a many-core system.
CHAPTER 8

Future work

At the end of every software project, there is room for additions to the program. First, both the front- and back-end have been designed in such way that respectively other back- and front-ends can be attached to them. It could, for instance, be interesting to attach ManyMan’s front-end to a different many-core chip. Also, for example, a light-weight text- or web-based interface could be created to be able to manage the SCC from machines with less graphical performance. A web-based interface would also add even higher portability to the system.

As stated in section 3.1.3, the Intel SCC provides functionality to dynamically change the frequency and voltage of cores and even allows users to individually turn them off. This functionality could of course be integrated into the tool, where it should be turned off in the front-end for chips that do not provide this type of control.

Currently, ManyMan can only display a task’s output as text. Probably, special output formats can be developed, using which, for example, line graphs, pie charts or images can be automatically generated from the output data. When this functionality would be implemented as a plug-in system, new formats could easily be added for specific tasks.

Some other improvements of the front-end include the addition of zoom levels. The user of the system may not always need to know what is happening on which core, but may only be interested in certain parts of the system. To provide this view, adding zoom functionality might be useful. Besides that, as discussed in section 5.3.5, adding a remote file browser could improve the task creation process.

The smart-start and smart-move functions can be improved, as discussed in section 5.1.2. For example, taking into account the performance history of the system or implementing any state of the art scheduling algorithm could result in smarter functions.

At the end of section 6.1.2, the possibility of writing the BLCR context files to RAM has been discussed. As cores do not share their RAM, a way to migrate these files between cores needs to be implemented. This could, for example, be done using copy cores [40] or memory remapping.

As the system can currently only be used on the university’s private network, no security had to be implemented. When one wants to use it somewhere else, some protection has to be built in. This includes, for example, password authentication or message encryption.
Finally, the current system has not been optimized for providing the best performance. Possible optimizations could be rewriting the back-end in C, or reading core and task performance from the /proc directory instead of using top. The performance of the communication layer could be improved by not using TCP and JSON, but using UDP and a custom optimized message format instead. Since the Kivy framework is used for the front-end, this has to be written in Python. This means that, other than changing Kivy’s source, there is no performance to gain here.


In order to use ManyMan with the Intel SCC, both the front- and the back-end that have been developed need to be started. In section A.1, the requirements and start-up instructions for the back-end will be discussed. Instructions on how to setup and use the front-end can be found in section A.2.

Sources of the entire system can be downloaded from [39]. These are free to use and modify, as ManyMan is released under a GPL 3 license [11].

A.1 Back-end

When one wants to use the back-end, one must have access to an Intel SCC chip. This chip should be running a 2.6.x Linux distribution and have the Berkely Lab Checkpoint/Restart library installed. A TCP port of the MCPC has to be remotely accessible to be able to communicate with a front-end. Finally, the MCPC should have a Python version newer than 2.6 installed. Note that Python 3 is not supported.

When all prerequisites are met, one has to configure the system as described in section B.1. Once this is done, the system may be started by navigating a shell to the path where the back-end is located and executing the following command:

```
python server.py
```

If the back-end needs to be stopped, this can be done by sending a SIGINT (Ctrl+C) signal to the shell in which the program is running. Please be patient as shutting down the system will also terminate all tasks started by the system. This might take some time.

A.2 Front-end

For using the front-end, one needs to have access to a machine with a monitor and either a multi-touch input device or a mouse. On this machine, a Kivy version newer than 1.2.0 needs to be installed. The machine should also have access a network, as it needs to communicate with a back-end.
ManyMan has to be configured as described in section B.2. When everything is set up, one has to navigate a shell to the front-end’s containing folder and execute the following command:

\[ \text{python main.py} \]

Instructions on how to use ManyMan can be found in section 5.3. When done, the visualization and management tool can be closed by pressing the Exit button in the upper right corner of the screen.
In order to be able to easily make changes to the way the programs run, configuration files are used. Both the back- and the front-end use the same syntax for these files. A basic configuration file could look like:

```json
messages:
[
  {
    stream: 'sys.stderr'
    message: 'Welcome'
    name: 'Harry'
  },
  {
    stream: 'sys.stdout'
    message: 'Welkom'
    name: 'Ruud'
  },
  {
    stream: $messages[0].stream
    message: 'Bienvenue'
    name: 'Yves'
  }
]
```

### B.1 Back-end

A configuration file for the back-end should contain at least the address information on which the program will run. If, for example, the back-end will run on local port 11111, the configuration file could look like:

```python
address: ['', 11111]
```

A complete list of all available configuration options can be found in table B.1.
### B.1 Complete list of back-end configuration options

<table>
<thead>
<tr>
<th>Option name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>list</td>
<td>List of [host, port] where the program will run.</td>
</tr>
<tr>
<td>dummy_mode</td>
<td>boolean</td>
<td>Indicator if the program will connect to the SCC cores or randomly generate data.</td>
</tr>
<tr>
<td>logging_format</td>
<td>string</td>
<td>Format of the log messages.</td>
</tr>
<tr>
<td>logging_datefmt</td>
<td>string</td>
<td>Date format of the log messages.</td>
</tr>
<tr>
<td>log_filename</td>
<td>string</td>
<td>Path to the log file.</td>
</tr>
<tr>
<td>logging_to_console</td>
<td>boolean</td>
<td>Indicator whether log messages should be printed in the console or not.</td>
</tr>
<tr>
<td>logging_level</td>
<td>string</td>
<td>Log level of the log file. Can be one of DEBUG, INFO, WARNING, CRITICAL.</td>
</tr>
<tr>
<td>logging_level_conso</td>
<td>string</td>
<td>Log level of the console. See above for possible values.</td>
</tr>
</tbody>
</table>

Table B.1: Complete list of back-end configuration options.

### B.2 Front-end

The configuration file for ManyMan’s front-end does not require a lot of information either. In this file, the address of the back-end and the path to the virtual keyboard directory should be present. The latter will in general be the `keyboards` folder in the directory containing the front-end. For example, a minimal front-end configuration file could look like:

```plaintext
address: ['sccsa.science.uva.nl', '11111']
keyboards_folder: 'C:/Users/Guest/Desktop/jimivdw/frontend/keyboards'
```

All available options for front-end configuration, along with their types and meanings are listed in table B.2.
<table>
<thead>
<tr>
<th>Option name</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kivy_version</td>
<td>string</td>
<td>Version of Kivy to use. Should be at least 1.2.0.</td>
</tr>
<tr>
<td>keyboards_folder</td>
<td>string</td>
<td>Folder containing the virtual keyboard file.</td>
</tr>
<tr>
<td>logging_level</td>
<td>string</td>
<td>Logging level of the application. Can be one of debug, info, warning, critical.</td>
</tr>
<tr>
<td>address</td>
<td>list</td>
<td>Address of the back-end in [host, port] notation.</td>
</tr>
<tr>
<td>framerate</td>
<td>float</td>
<td>Framerate at which the core overlay will be updated.</td>
</tr>
<tr>
<td>bufsize</td>
<td>integer</td>
<td>Size of the TCP message buffer.</td>
</tr>
<tr>
<td>tasks</td>
<td>list</td>
<td>List of initial tasks on the right side of the screen. A task consists of a command and optionally a name, e.g. {name: 'pi', command: '/shared/jimivdw/jimivdw/tests/pi'}.</td>
</tr>
<tr>
<td>core_background</td>
<td>string</td>
<td>Location of the core image.</td>
</tr>
<tr>
<td>core_background_active</td>
<td>string</td>
<td>Location of the active core image.</td>
</tr>
<tr>
<td>core_border</td>
<td>list</td>
<td>Size of the core image’s border in [top, right, bottom, left] format.</td>
</tr>
<tr>
<td>core_padding</td>
<td>integer</td>
<td>Padding between core images.</td>
</tr>
<tr>
<td>core_color_range</td>
<td>list</td>
<td>Range of payload colors in [no_usage, full_usage] format.</td>
</tr>
<tr>
<td>task_default_color</td>
<td>float</td>
<td>Hue value of the tasks in the pending list.</td>
</tr>
<tr>
<td>task_info_image</td>
<td>string</td>
<td>Location of the task info image.</td>
</tr>
<tr>
<td>task_dup_image</td>
<td>string</td>
<td>Location of the task duplicate image.</td>
</tr>
<tr>
<td>task_start_image</td>
<td>string</td>
<td>Location of the task start image.</td>
</tr>
<tr>
<td>task_stop_image</td>
<td>string</td>
<td>Location of the task stop image.</td>
</tr>
<tr>
<td>task_pause_image</td>
<td>string</td>
<td>Location of the task pause image.</td>
</tr>
<tr>
<td>task_resume_image</td>
<td>string</td>
<td>Location of the task resume image.</td>
</tr>
<tr>
<td>task_move_image</td>
<td>string</td>
<td>Location of the task move image.</td>
</tr>
<tr>
<td>output_buffer_size</td>
<td>int</td>
<td>Number of lines of output to show in the detailed task view.</td>
</tr>
<tr>
<td>perfgraph_default_history</td>
<td>string</td>
<td>Number of timesteps to show in the performance graphs.</td>
</tr>
</tbody>
</table>

Table B.2: Complete list of front-end configuration options.
Communication protocol

All messages in the system have the same base-format:

```json
{
  "type": <type>,
  "content": {
    <content>
  }
}
```

Using this format, messages can easily be identified and handled.

Before any other message can be sent, the front-end has to identify itself to the back-end by sending a `client_init` message. This could, for example, look like:

```json
{
  "type": "client_init",
  "content": {
    "name": "ManyMan"
  }
}
```

When the message is of a correct format, the server will reply by sending its `server_init` message:

```json
{
  "type": "server_init",
  "content": {
    "name": "Intel SCC",
    "cores": 48,
    "orientation": [
      [37, 39, 41, 43, 45, 47],
      [36, 38, 40, 42, 44, 46],
      [25, 27, 29, 31, 33, 35],
      [24, 26, 28, 30, 32, 34]
    ]
  }
}
```
If the front-end receives this message correctly, all other messages can be sent. For the front-end, these messages are:

- **task_start**
  - `name`: name of the program.
  - `program`: command that has to be executed.
  - `core*`: optional core on which the task should run.

- **task_move**
  - `id`: ID of the task.
  - `to_core*`: optional core to which the task should be moved.

- **task_pause, task_resume, task_stop, task_duplicate**
  - `id`: ID of the task.

- **task_output_request**
  - `id`: ID of the task.
  - `offset*`: optional offset in the output.

The back-end can now send the following messages:

- **status**
  - `chip`
    - `cores`: list of cores and their payloads.
    - `tasks`: list of tasks and their payloads.

- **task_output**
  - `id`: ID of the task.
  - `output`: output of the task.

- **invalid_message**
  - `message`: the reason why this message has been sent.
Usability test

Please execute the tasks listed below. Any remarks can be noted per task.

1. Make two copies of the "Pi" task.

2. Start one of the copies of "Pi" on core 20.

3. Start the "Sleepy greeter" task on core 20 as well.

4. Open core 20’s detail view.

5. Look at "Sleepy greeter”’s output.

6. Close all opened popups (detail views). Create a task called "Test” with command ”/shared/jimivdw/jimivdw/tests/mem 10 1000”.

7. Start the just created "Test” task on core 30. Also start one of the "Pi” tasks on core 31 and a "Memory” task on core 32.

8. Open the detail views of all cores that have been used in the previous step and arrange them in such way that the performances of each of the cores are easily comparable.

9. Move the "Test” task to core 32.
10. Stop the "Memory" task.

11. *Smart-start* the last "Pi" task.

12. Pause this just started task, wait 5 to 10 seconds and resume it.

13. Start the "Count" task on core 47, look at its output and *smart-move* it.

14. Create four copies of the stopped "Memory" task and restart all of them (including the original) on any core you want.

Finally, please answer the following questions:

1. Is the process of starting tasks intuitively?

2. Is the process of moving tasks intuitively?

3. Is the payload of the entire SCC system visualized in a clear way?

4. Is the payload per core / task visualized in a clear way?

5. Are the used icons well chosen?

6. Do you have any other remarks?

Thank you for participating in this experiment!