Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis
at the Department of Information Technology
and Electrical Engineering

Designing a benchmark for many-core architectures

AS 2012

Pierre Ferry

Advisors: Devendra Rai
Lars Schor
Professor: Prof. Dr. Lothar Thiele

Zurich
10th February 2013

Abstract

Recent progress in computing capabilities is no more driven by the increase of processor frequency, but rather by an increase in the number of cores. In this context, it is necessary to distribute tasks on a given multi/many- core system in a way that the combined computing power of the entire system can be most efficiently harvested.

Benchmarks are an efficient tool to clearly understand the computing and communication capabilities offered by the multi/many core system. Once a clear understanding of various available resources has been developed, it is possible to develop algorithms that will map given tasks to the multi/many-core system such that various metrics such as latencies are minimized. This thesis proposes a comparative study of existing parallel applications and adapts one of them, a deduplication algorithm, for a many-core architecture using the Distributed Application Layer (DAL) framework. The benchmarks were evaluated on a PC as well as the Intel SCC cluster, and detailed results are presented in this thesis.

Acknowledgements

I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for granting me the opportunity to write this semester thesis in his research group.
I would also like to thank my advisors Devendra Rai and Lars Schor for their support and availability throughout this thesis.

Contents

1 Introduction **2**
- 1.1 Motivation . . . 2
- 1.2 Contributions . . . 3
- 1.3 Outline . . . 4

2 The KPN and DAL **5**
- 2.1 The Kahn Process Network, KPN . . . 5
- 2.2 The Distributed Application Layer, DAL . . . 6

3 Benchmark Development **8**
- 3.1 Finding the benchmark . . . 8
- 3.2 The Deduplication application . . . 11

4 Benchmark **15**
- 4.1 Measurement libraries . . . 15
  - 4.1.1 Performance Measurement on a PC-Cluster . . . 15
  - 4.1.2 Performance Measurement on the Intel SCC . . . 17
- 4.2 Benchmark Results . . . 19
  - 4.2.1 PC Cluster . . . 19
  - 4.2.2 Intel SCC . . . 23

5 Conclusion and Outlook **27**
- 5.1 Conclusion . . . 27
- 5.2 Outlook . . . 27

A Acronyms **29**

B Presentation Slides **30**

List of Figures

3.1 Dedup, Process network of the deduplication application. . . 12

4.1 Communication output of each process as a function of the input file size. 20
4.2 Number of hardware cycles spent in computation for each process as a function of the input file size. 21
4.3 Average spent hardware cycles per byte. 22
4.4 SCC Layout. 23
4.5 Average computation time by process as function of input size. 24
4.6 Average computation time by process as function of input size. 25
4.7 Average computation time for compression with duplicated input. 26

1 Introduction

1.1 Motivation

The goal of this project was to design and to implement a benchmark for many-core architectures. Developing a benchmark for a many-core system is much more complex than developing a corresponding benchmark for a single-core architecture. Usually, for a single core system one can have a rather accurate idea of a processor's overall performance with its memory and computational performance. Performance depends on a great variety of parameters such as clock frequency, instruction set, cache size. Once these parameters are known, it is rather easy to assert how well this processor will perform a specific task.

However, a many core system is usually more complex than a single-core system. A multi/many core system may have complex cache and memory hierarchy. First of all, the core hierarchy may not be homogeneous, with only some cores sharing a given cache, while others may map to entirely different memory regions. It is also possible that a given multi/many-core system may have cores of different types: for instance, some cores may be x86-based cores, while others may be graphics processors. Such heterogeneous cores, together with complex memory hierarchies may generate very diverse memory access patterns that makes it difficult to evaluate the memory performance. A given application may therefore perform differently depending on how it is mapped onto the architecture. In view of such challenges, it becomes clear that

mapping a significantly large application onto a multi/many-core system is a complex problem. Common approaches used to solve such mapping problems rely on design-space exploration (DSE), wherein abstract properties (such as maximum computing power of a core, available bandwidth to its neighbours) of the given system are used to construct a suitable model of the given system. Suitable algorithms are developed, which use such abstract models in order to arrive at an optimal solution to, for instance, map various processes in the application such as the communication latencies between processes are minimized. The more cores are available and the more complex the application is , the more difficult it is to evaluate the performance of an application on a predefined architecture.
A benchmark gives a reliable way to compare different architectures and application mappings.

1.2 Contributions

In this thesis, a set of existing parallel benchmarks was studied to find an application fitted for a high performance system benchmark. These benchmarks were compared regarding a few parameters:

- Parallelizability
- Possibility to scale computation and communication load to match more or less performant systems
- Complexity
- Realistic as a nowadays task for high performance systems

After a careful examination two benchmark applications were developed:

- *Secure Hash Algorithm (SHA) signature* application : relatively simple application that computes secure signature of files. It is used to authenticate file transfers for instance.
- *Deduplication* application: detects and removes redundancy in a file and compresses it. This application will be the one described in this report as it is the one of significant importance.

Instrumentation to measure the performance of the system was integrated to the application. The resulting benchmark was run on a PC cluster and ported on an Intel Single-chip Cloud Computer (SCC) (see 4.2.2).

1.3 Outline

In the first chapter, a computation model, Kahn process networks, is presented as well as the Distributed Application Layer.
In the second chapter, the steps leading to the benchmark development are described. First a comparative study of existing parallel benchmarks is summarized. Then the *Deduplication* application principle is described as well as its implementation.
The third chapter presents the solution used to measure the performance of the application as well as the results of the benchmark on a PC cluster and on an Intel SCC chip
The last chapter stands as a conclusion and an Outlook to this thesis.

2

The KPN and DAL

2.1 The Kahn Process Network, KPN

A Kahn Process Network (KPN) is a model of computation that was proposed in 1974 by Gilles Kahn (see [1]) . In this model, a KPN is an oriented graph where nodes are sequential processes and edges are communication channels (unbounded First In First Out buffers). Processes can either write or read from these channels. They must also have the following properties:

- Writing to a channel is *non-blocking*: every `write` is successful and the process does not stop.
- Reading from a channel is *blocking*: the process stalls until data is available in the channel.
- Monotonicity: with a prefix of any input sequence, the process produces a prefix of the corresponding output sequence.

These properties, alongside with the structure of the network guarantee the determinism of the application. The monotonicity also allows processes to run asynchronously or in any given order since a process does not need to have access to its full input sequence to produce the beginning of the output sequence.
It is important to note that processes can not test the state of the channels (the availability of data) because it would make the execution of the process timing dependent.

In the case of our application, the restriction on the `read` primitive was removed in order to ensure the maximum usage of resources (minimum possible process stalling). Thus, non-blocking read have been allowed. This change removes the guarantee of determinacy of the process network. However, in the particular case of our application, the behavior of the application remains correct even without this restriction (see 3.1).

2.2 The Distributed Application Layer, DAL

The Distributed Application Layer (DAL) is a framework developed by the Computer Engineering and Networks Laboratory (TIK) at ETH. It allows to map KPN-like process networks (see 2.1) dynamically onto many-tile platforms. Thus, mapping can be changed at execution time in order to adapt to the current application scenario.

The DAL framework makes it possible to use the same application on different multiprocessor architectures because the application is designed without any knowledge of the architecture on which it will be executed. The application mapping is described separately from the application itself.

In this framework, an application is designed as a network of processes. Every process has the same three methods: *init*, *fire* and *finish*. The *init* method is executed by every process at the start of the application. This procedure may be used, for example, to initialize a data-structure needed for the application.

The *fire* method is called repeatedly, until the application stops executing. It is the *fire* method which executes the functionality of each process. Thus, all necessary computations and communications are carried out within the *fire* method itself.

The last one, *finish*, is called when the application stops, and may be used to perform necessary housekeeping activities, such as freeing `malloc'd` memory, write results to a filesystem, or to collect relevant performance statistics of the process.

It is possible for a process to have access to *persistent* memory, in order to share certain variables, or to retain the results of computation between successive *fire* cycles. Such persistence of information is made possible by the use of so-called `Local_State` of the process.

The DAL Framework is composed of two layers: A bare-bones framework, which creates a *master* process and a set of *slave* processes. The *master* processes dispatches instructions to the required *slave* process, such as:

1. Load a certain functionality for execution.
2. Create a FIFO of a given size, and attach it to *either* the input or an output port of a process.
3. Instantiate an inter-processor communication module, and attach it to a given FIFO.
4. Start, Stop, Pause or Resume the execution of a given process.

In order to keep the memory footprint of the framework as low as possible, the functionality to be loaded by a slave process is compiled a so-called *shared library, or process-independent code (PIC)*. At run-time, a slave may receive an instruction to load the functionality associated with *Reorder* process of the *Deduplication* application. The slave simply looks up a library consisting of a collection of shared-objects, and loads a suitable object. In what follows, the functionality will be simply referred to as a *process*, not to be confused with the *master* or the *slave* processes. Wherever necessary, the distinction will be made clear with the context.

All *slave* processes continuously listen for instructions from the *master* process. A process network description file, written in XML, is created by the designer of the application which describes how various processes in the application are connected, alongwith the necessary size of associated FIFOs for communication. Thus, the *master* process derives necessary application setup instructions from this process network description file.

Upon reception of these instructions, the *slave* processes establish the corresponding channels and load the process specified by the *master* process.

When the process network has been established, the *master* will start execution of the application, by sending the appropriate instruction(s) to each of the *slave* processes. Subsequently, the processes of the application run asynchronously to each other. The communication layer is established using the Message Passing Interface (MPI) [2]. Since message passing interface is the most commonly used communication solution used in high-performance computing applications, versions of it have been optimized for use on various multi/many- core systems. Such commonality of communication interfaces allows the application to be portable across several multiprocessor architectures.

This allows us to develop our application as a KPN-like process network and run it on both on a PC cluster and an Intel SCC.

3

Benchmark Development

3.1 Finding the benchmark

An ideal benchmark for our purpose must be based on a realistic application, and must offer computational and communication loads which are commonly expected of high-performance applications today. Another interesting characteristic that we look for in a benchmark is that it should show a deterministic relationship between the size of its input, to communication and computational work done by it. Ideally, we look for a polynomial relationship between the input size and output loads (communication, computation).

A benchmark possessing these characteristics can be tuned to test a range of multiprocessing systems (mobile platforms, Intel SCC, or the supercomputing cluster). For instance, each core in the Intel SCC chip is a relatively simple P54C core, running at 800MHz, with 16K of instruction and data cache, and therefore, the computing capabilities offered by each core of the SCC are significantly different from those offered by a core in modern cluster systems, which may be running latest Intel/AMD cores, clocked at 3GHz or higher.

The first step was therefore to find an application that could make a suitable benchmark for this thesis. An important point is that, in the DAL framework, processes do not share any memory and so the target application must be parallelizable with explicit communication between processes. There

are a lot of existing benchmarks for parallel computing but most of them rely on shared memory (i.e., thread level parallelization). Aside from being suitable for a distributed architecture, the benchmark indeed needs to be reasonably complex in order to be able to exploit a many core system but also to be feasible in the amount of time available for this thesis.

Some of the compared benchmarks or set of benchmarks that were studied are described in the following table:

Benchmark	Source Language	Parallelism	Complexity	Stresses	Remarks
NASbench	Fortran, Java, OpenMP, MPI	Parallelized, (thread-level,OpenMP)	Quite complex	Varies according to selected benchmark	Based on computational fluid mechanics algorithms, source not comment enough
H264	C	Complex parallelism, implementation dependent parallelization strategy	Quite complex	Computation and Memory	Signal Processing Algorithms. Source not commented enough.
Parmibench	C	Master/slave based strategy	Simple to Medium	Computation and Memory	Search, Hash, Optimization algorithms
PARSEC	C	Thread-based parallelization (POSIX threads)	Varies	Varies according to selected benchmark	Computation, Recognition, Encoding/Decoding, Data mining algorithms

Table 3.1: Comparison of various benchmark candidates

The NASA benchmarks are a set of benchmarks developed by the NASA to evaluate parallel systems [3]. Most of these benchmarks are computational fluid mechanics applications that would theoretically fit our expectations. These benchmarks stress the system in different ways (computation, I/O, memory) . However the source code of these benchmarks is often only available in Fortran or Java and the parallelism is introduced using the OpenMP library, which makes such benchmarks applicable to only shared memory systems. The sources are not well-commented, and there exists very little documentation which makes it very difficult to adapt these set of benchmarks for our purposes.

The H264 encoder/decoder was a strong candidate benchmark application for this thesis. H264 is an advanced standard for video compression, and is in

wide use today, see [4]. For example, this standard is used to compress videos for the blue-ray disc format. Real-world applications, and non-trivial computational requirements make the H264 an interesting benchmark. Further, a distributed implementation allows evaluating video compression speeds and related metrics on a variety of multicore and unicore platforms.

However a few problems make it difficult to use this benchmark for the current thesis. First, the method to parallelize the application is not entirely clear, many different approaches exist [5][6][7]. Parallelism can be introduced at different levels of the computation. Frame processing or macro-block processing are two steps that can be parallelized (but not both at the same time). Each choice having its benefits and drawbacks. These techniques are also not indefinitely scalable, the number of worker being limited by the complex dependency between the encoding of different frames/blocks. Moreover both the encoder and the decoder are very complex and adapting one of them in the course of the thesis was not really feasible.

Parmibench is a set of parallel benchmarks developed by [8] . Most of these benchmarks are in fact quite simple algorithms which are commonly used. The code source is fully available. However the parallelization technique used is not very interesting for our benchmarks: it is mainly done by a master process that segments data and distributes it to workers that are all alike.

PARSEC [9] is the last set of benchmarks that will be described here. It was developed at Princeton University's Computer Science Department. The entire source code of all benchmarks in the suite is available. The benchmarks proposed in this suite are of very varying complexity. The simplest one computes the price of financial options according to the Blackscholes formula [10], and is commonly used in financial applications. This application is interesting because the speed of computations can have a major impact on the financial markets. However the *blackscholes* algorithm is too simple to be used as a benchmark. Another interesting benchmark in the PARSEC suite is called *Ferret* and is used to find images based on content similarity. *Ferret* has applications in comparing, for instance, a database of fingerprints to a sample, or image-search (such as Google's image-search) where the search engine tries to find images similar to the one specified as an input. *Ferret* is reasonably complex, and could be very well adapted to our framework. However it needs to be interfaced with a specific database that would be difficult to integrate in our application and would hurt its portability.

The benchmark that was finally chosen for this thesis is part of PARSEC. It is a *deduplication* algorithm. This program processes a stream of data and tries to detect redundancy in the data stream. If any redundancy is detected,

deduplication attempts to remove it, without loosing any information, and then attempts to compress this data. Thus, *deduplication* is useful for use in data centers, allowing storage space to be freed by avoiding to storage of redundant information [11].
This application is very interesting because it can be easily split into different steps, and can be also very easily parallelized. It can also be used on working set of arbitrary size so that it is possible to scale the input to match the kind of system that has to be benchmarked.

3.2 The Deduplication application

The *Deduplication* application is used to remove redundancy in an input stream. The application fragments input data and recognizes identical segments in it. Non duplicated parts are compressed into the output whereas simple references are written for the duplicated parts.

The application consists of six different processes. Fig 3.1 shows the network for a two pipelines architecture. The data is split by the application in chunks. Processes then only communicate these chunks that are processed step by step as they go through the application.

The C structure of a chunk is the following :

```
typedef struct _chunk_t {
  struct {
    int isDuplicate;        //whether this is an original chunk or a duplicate
    chunk_state_t state;    //which type of data this chunk contains
  } header;
  //The SHA1 sum of the chunk, computed by SHA1/Routing stage from the uncompressed chunk
data
  unsigned int sha1[SHA1_LEN/sizeof(unsigned int)];
  //The uncompressed version of the chunk, created by chunking stage(s)
  mbuffer_t uncompressed_data;
  //The compressed version of the chunk, created by compression stage
  //based on uncompressed version (only if !isDuplicate)
  mbuffer_t compressed_data;
  //reference to original chunk with compressed data (only if isDuplicate)
  struct _chunk_t *compressed_data_ref;

  //Original location of the chunk in input stream (for reordering)
  sequence_t sequence;
  //whether this is the last L2 chunk for the given L1 number
  int isLastL2Chunk;

} chunk_t;
```

This structure contains information about the chunk: its number to be able to reorder the chunks at the end of processing, whether the chunk is duplicated or not (this variable is set only after the dedup step), the SHA signature

of the chunk and the pointer to the data of the chunk. These chunks are sent through three separate channels: one that sends the chunk meta data, e.g the previous structure, the second one sends the size of the data and the last one sends the actual data. It is necessary to use a separate channel to communicate the size of the data since the MPI receiver needs to know how many bytes it has to read.

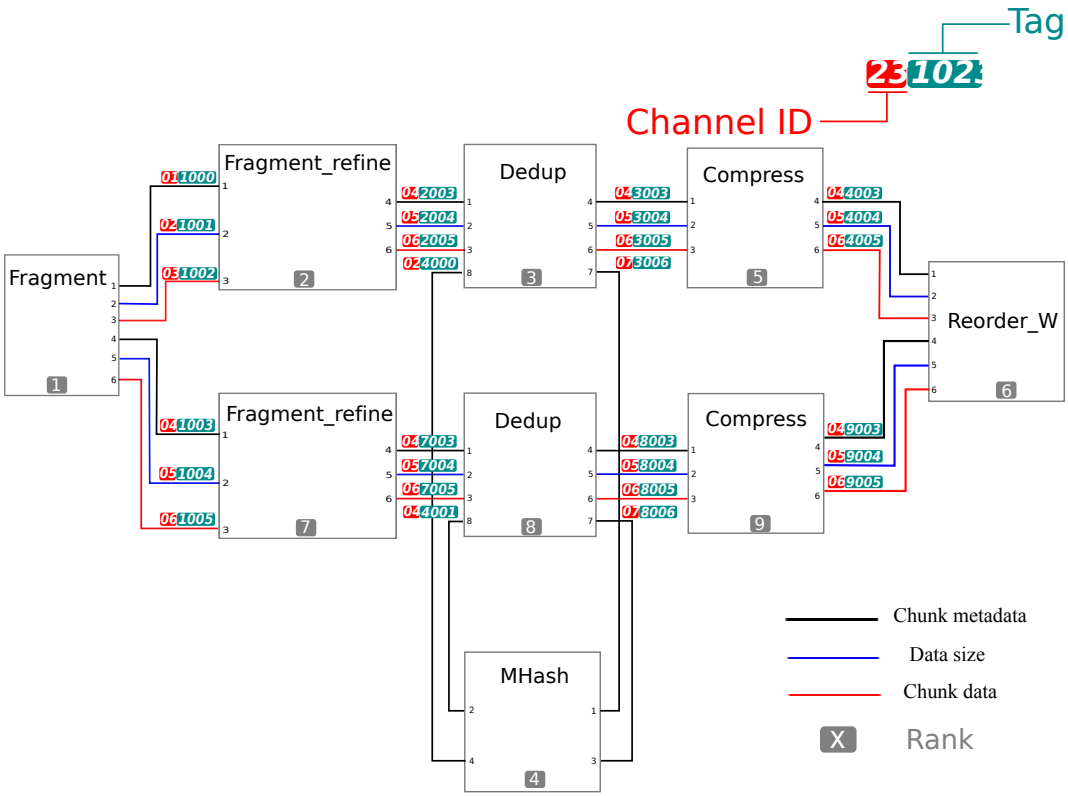


Figure 3.1: Dedup, Process network of the deduplication application.

The first two processes are `Fragment` and `Fragment_refine`. The role of these processes is to :

- Read data
- Split it in chunks
- Annotate them to keep track of their order

To split data, these processes try to find specific breakpoints using rabin fingerprints [12].

A data window is considered as a polynomial over the two elements finite field: if $m_0, m_1 ... m_{n-1}$ is the data, we consider the polynomial $m_0 + m_1 x^1 + ... + m_{n-1} x^{n-1}$. The fingerprint is then the remainder of the division of this polynomial by a random irreducible polynomial. A data point is considered as a splitpoint if the k-lowest bits of the fingerprint equal zero.
It can be proven that, since the division is done by an irreducible polynomial, the fingerprints are pseudo random and therefore the probability of this point to be a split point is 2^{-k}. The advantage of this method over another hash function is that it doesn't need to do the full computation for every window but can use the result of the previous data window.
This process is done in two steps to avoid to bottleneck the application. The goal of the first process is indeed to read the input file and to begin to split the data in chunks. The reading and splitting cannot be parallelized if there is only one data source. Hence it makes a coarse fragmentation of the data by making big jumps in data and only trying to find a breakpoint rather far from the previous one. The coarse chunks resulting from this first fragmentation are then labelled and dispatched alternatively to the different `Fragment_refine` processes.

In `fragment_refine` data is further fragmented and every breakpoint is found. The size of chunks can be very diverse, as well a few hundred bytes as more than ten kilobytes. Chunks are then sent to the `Dedup` processes where the actual *deduplication* will be done.

The role of the `Dedup` process is to check whether an incoming chunk has been encountered before. To do so, it computes the SHA signature of the chunk and checks whether it has an entry in its hashtable. If not, the signature is added to the table and the chunk is sent to the next process. The main issue with this system is that it can't really be parallelized since all the `Dedup` processes must have knowledge of all the already seen chunks. This is usually not an issue with shared memory systems where a common hashtable can be used. To address this issue, a system similar to cache has been established. Each `Dedup` process has a local hashtable to keep track of the chunks it has personally encountered and a common hashtable hosted by another process `Mhash`. In case of cache miss, if a chunk has not been seen before by `Dedup`, it makes a request to `Mhash`. The latter then answers whether it has already been seen or not and adds it to its table if it has not. This means that `Mhash` can once again be a bottleneck for the application, because it has to handle all the application branches. However the amount of computation it has to do is minimal, only a hash query, and it does not need to communicate big amounts of data, only the metadata of chunks which is only a 104 Bytes load compared to the potential tens of kilobytes of a chunk. It should therefore be able to handle a good number of branches

before actually slowing down the application.
The metadata of the chunk is annotated depending on whether it is a duplicate or not. The chunk can then be sent to the next process, `Compress`.

The role of compress is simply to compress the incoming chunk in order to spare a maximum of space. The compression method that was chosen here is a simple zip compression. A chunk is however compressed only if it is not a duplicate. The compressed (or not compressed if not necessary) data is then sent to the last process, `Reorder_W`.

The last process reorders the chunks that do not necessarily come in order because of the different branches of the application. When a chunk is read by the process, the process checks whether it is the next to be written depending on what is its sequence and what was the previous chunk written. If it is not the next chunk, it is put in a search tree. If it is the next chunk, it is written in the output file and the process checks if the next one is also in the search tree. Once all the chunks available are written, the process tries to read a new chunk again.
This process writes the compressed data in the output file if the chunk is not duplicated and only writes its SHA signature if the chunk is duplicated.

It can be noted that with this architecture it is possible to obtain in the output file a SHA signature before the original has been written. Two identical chunks can indeed be processed out of order and in that case the last will be treated as an original whereas the first will be considered a duplicate. For the first one only a signature will be written and the next occurrence will be compressed. This possibility is not incorrect but must be taken into account by the corresponding decoder.

A few key parameters are of note for the configuration of the application:

- `MAXBUF` : size of input buffer loaded at a time by `Fragment`. This variable is defined in *dedupdef.h*.
- `ANCHOR_JUMP` : minimum size of the coarse chunk defined by `Fragment`. This variable is defined in *dedupdef.h*
- `rabinmask` : defines the value that the lower bits of the rabin fingerprints have to take to define a breakpoint. This variable is defined in *rabin.h*

4 Benchmark

4.1 Measurement libraries

Once a benchmark is adopted for use in the DAL framework, the essential step of introducing appropriate instrumentation in order to extract various performance parameters must be completed. The instrumentation must be carefully done in order to collect accurate information about various metrics of interest (computation times, communication size) as each process executes.

The instrumentation step was carried out both in the PC-cluster environment, as well as on the Intel SCC. Such an arrangement allows us to cross-compare the results from both environments, and derive conclusions about capabilities of both setups.

4.1.1 Performance Measurement on a PC-Cluster

Timing

For the case of a PC-cluster, *Performance API, (`PAPI`)* library ([13]) was used. This library utilizes hardware-counters in the processor to keep an accurate track of a range of events as well as time. `PAPI`'s application programming interface allows a user to select the events of interest. Once a selection of interesting events to monitor has been made, the library provides functions to access values recorded in hardware counters (such as number

of L1 cache misses, between two known points in the software code). The counters used for the PC-cluster were the following:

- TotalCycles : Number of machine cycles elapsed between two points of the process.
- Instructions completed : Number of instruction completed by the process.
- Hardware Interrupts : Number of times the process has been interrupted by the Operating System (OS).
- L1 Data Cache Misses : Number of cache misses of the first level cache.
- Real Time : Time elapsed between two points of the process.

The process' source code is annotated with calls to initialize the `PAPI` library, and then to execute the event-measurements between two points in the process' source code. The starting point of measurement is delineated with a call to `Initialize_Performance_Library(...)`.The end point of measurement is denoted with a call to `Finalize_Performance_Library(...)`. The actual definitions of `initialize_` and `finalize_` follow.

The following method is used to start counters :

```
void Initialize_Performance_Library(int *EventSet, struct timeval* tv_start)
{

    int retval;
              /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);

    if (retval != PAPI_VER_CURRENT) {
      fprintf(stderr, "PAPI library init error!\n");
      exit(1);
    }
    gettimeofday(tv_start,NULL);
    /* Start counting */
    if (PAPI_start_counters(EventSet, NUM_EVENTS) != PAPI_OK)
    printf ("%s:%d\t ERROR\n", __FILE__, __LINE__);
    printf("PerformanceLibrary Initialized, EventSet Address: %p\n", EventSet);
fflush(stdout);
}
```

The following method is used to stop counters and record results:

```
void Finalize_Performance_Library(int *EventSet, struct timeval* tv_start, struct
timeval* tv_stop, long long* values, FILE* fp)
{

      gettimeofday(tv_stop,NULL);
      if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK)
      {
       printf ("%s:%d\t ERROR\n", __FILE__, __LINE__);
      }

      fprintf(fp, "%lld\t%lld\t%lld\t%lld\t%1.9f\t\n",*((long long*)values),*((long
long*)values+1), *((long long*)values+2), *((long long*)values+3),(((double)(tv_stop-
>tv_sec*1000) + ((double)tv_stop->tv_usec)/1000) - ((double)(tv_start->tv_sec*1000) +
((double)tv_start->tv_usec)/1000)));fflush(fp);
      printf("PerformanceLibrary Finalized, EventAddress: %p\n", EventSet);
fflush(stdout);
}
```

When measuring process' computation times, calls to `Initialize_Performance_Library` and `Finalize_Performance_Library` were carefully placed so as to exclude any time spent for communication. Since a process can have any number of such so-called *compute-segments*, the total over all such compute segments is calculated.

Communication load

To measure the communication load of each process, the amount of sent bytes can simply be recorded at each write primitive and printed.

Memory

To measure the memory usage of each process, the standard `getrusage` function has been used. This function maintains a data-structure to keep track of a few statistics about a process and in particular the one that interests us: the maximum amount of memory used by the process, see [14]. According to the documentation, the data-structure member `long int ru_maxrss` indicates the maximum resident size of the process, in kilobytes.

4.1.2 Performance Measurement on the Intel SCC

The application was run on Intel SCC (see 4.2.2) in order to obtain reliable timing informations. Indeed this chips allows to run the application on baremetal, meaning that there is no OS running on the chip at the same time. In these conditions, the application is the only program running on

the chip. There are no interferences with other programs that could cause interruptions. There is also no other application using the cache memory that could modify the behavior of the application. Processes also run on separate tiles on the chip (see 4.2.2), so that we can be assured that there are no cache interference between different processes of the application (If two processes run on the same tile, they both use some common cache).
Simple recordings of the clock cycles of the chip give precise and reliable timing information about the execution of the application.

4.2 Benchmark Results

4.2.1 PC Cluster

The first set of numbers that have been measured have been on a PC cluster. It is not possible to get reliable timing on a PC cluster. First the communication is quite slow compared to communication between processors on a single chip. Also these PC are running many different threads at the same time and so depending on how busy the PC is during execution, the benchmark might give different results. But other metrics are still valid. Therefore only communication load, memory usage and number of hardware cycles spent by the process have been measured.

Communication load as well as machine cycles should evolve in a rather linear fashion as functions of the input file size. Experiments have been conducted with different input file size up to 1MB. Bigger input files are of course possible. However the speed of the PC cluster communications was such that a 10 MB input would take a long time to complete.

One can observe in fig 4.1 that the amount of data communicated by every process evolves linearly with the input file size. This is what one would expect since every process has to go through data in a linear fashion and that the number of chunks is linear with the size of the input file. It is also possible to see that the amount of communication is much lower for the `Mhash` process simply because it only forwards the chunks meta data that is about a hundred bytes big.

The difference between two similar process (for instance between `Dedup1` and `Dedup2`) is due to the granularity of the load repartition. As explained before, the first process makes big chunks (that can be as big as our architecture allows it) and distributes it to the second layer of processes. The default settings of the original benchmark was of 2MB. For the purpose of this study, it was greatly reduced (to about 30 kB) to be able to distribute data across both the application branches. It still gives significative difference between the branch that was served first and the other one (it can also be an almost zero difference if they have been served the same number of times).

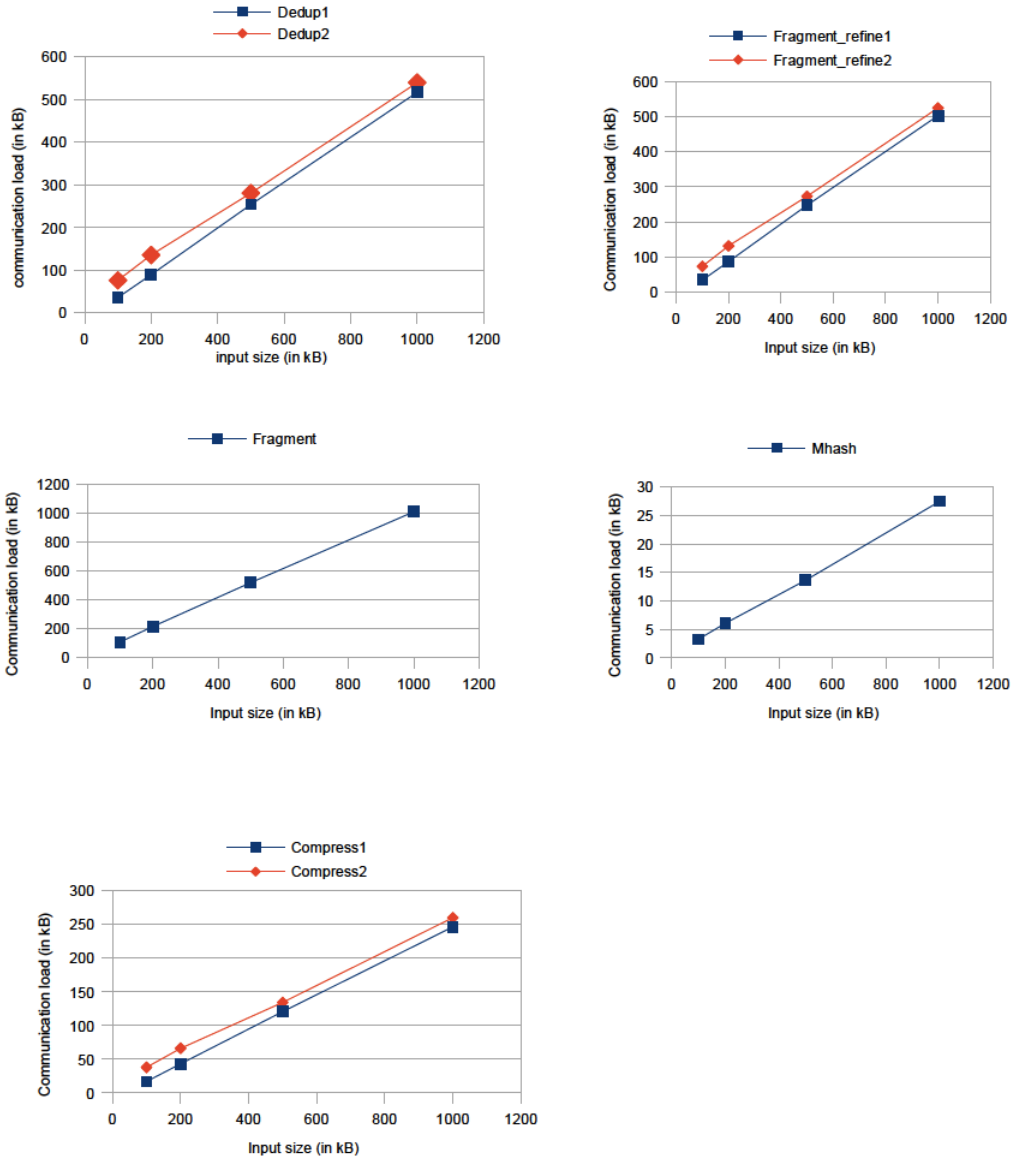


Figure 4.1: Communication output of each process as a function of the input file size.

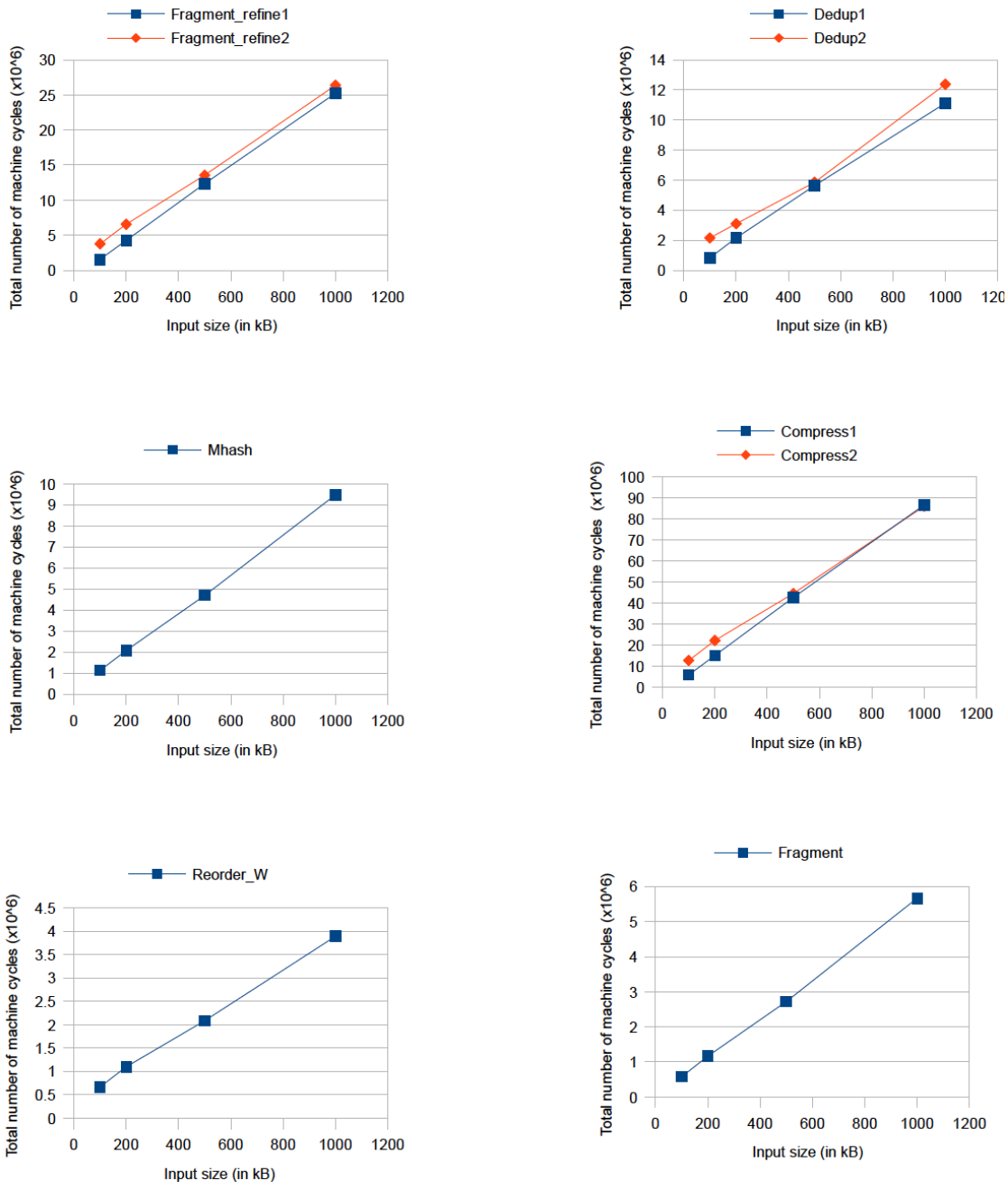


Figure 4.2: Number of hardware cycles spent in computation for each process as a function of the input file size.

Figure 4.2 shows that the number of hardware cycles varies as a function of input file size in a very similar fashion to the communication load. It is

relatively natural because the amount of cycle spent by chunk is constant in average. These measurements are based on single runs.

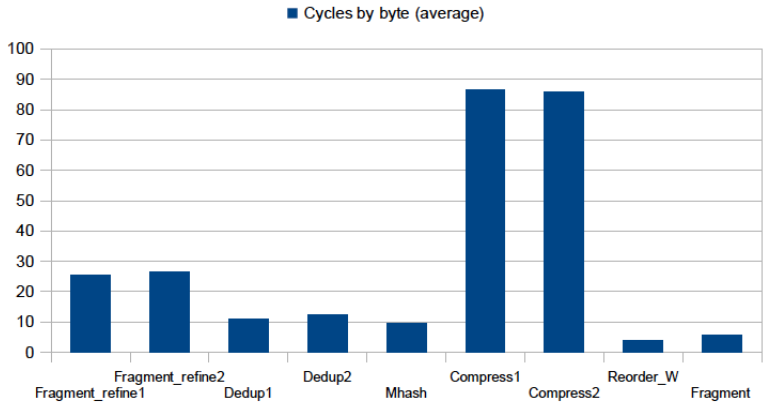


Figure 4.3: Average spent hardware cycles per byte.

Figure 4.3 shows the number of cycles the application needs in average to compute one byte of data. This graph shows that the compression stage is by far the one that takes the most computation. Then comes in second the fine fragmentation. Third is the deduplication (because of the SHA signature calculation). Last are the non-parallelized processes. These confirms that further parallelization should be possible. It is also possible to further reduce the computation time per byte for `Fragment` by making bigger chunks at the first stage. There are also solutions to parallelize `Mhash` if it becomes really necessary.

4.2.2 Intel SCC

In order to obtain measurements on timings that are reliable, the application was ported onto a Intel SCC. The Intel SCC is a many-core processor containing a total of 48 cores organized in a 6x4 on-die mesh of tiles with two cores per tile[15]. Each tile has a small shared memory area with the rest of the chip called Message Passing Buffer (MPB) that is used to improve performance of communication between cores. Routers as shown on 4.4 form a 2D-mesh communication network that guarantee communication between tiles. Four memory controllers (MC), available in each quadrant of the SCC allow each core to access an external large memory. For the experiments, the Intel SCC was operated with the following parameters:

1. *Core Clock Frequency*:800MHz
2. *Router Frequency*:1600MHz

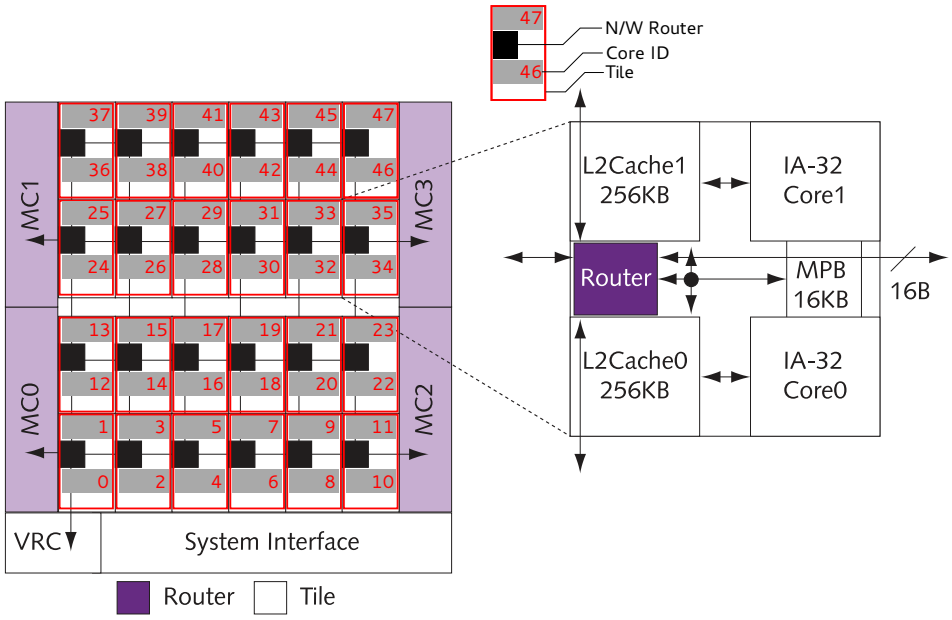


Figure 4.4: SCC Layout.

There is no Operating System (OS) loaded onto the SCC, the only process running on the chip when the application is running is the process of the application. Therefore we can obtain accurate timing measurements. The computation time measurements were repeated between 5 to 10 times each

on two different inputs. Figures 4.5 and 4.6 show the average value of computation time. The span of results is very small which shows that the amount of time required to execute a specific task is very stable. The first input is a sequence of random inputs so it does not contain any duplicate. The second input however contains a lot of duplicates.

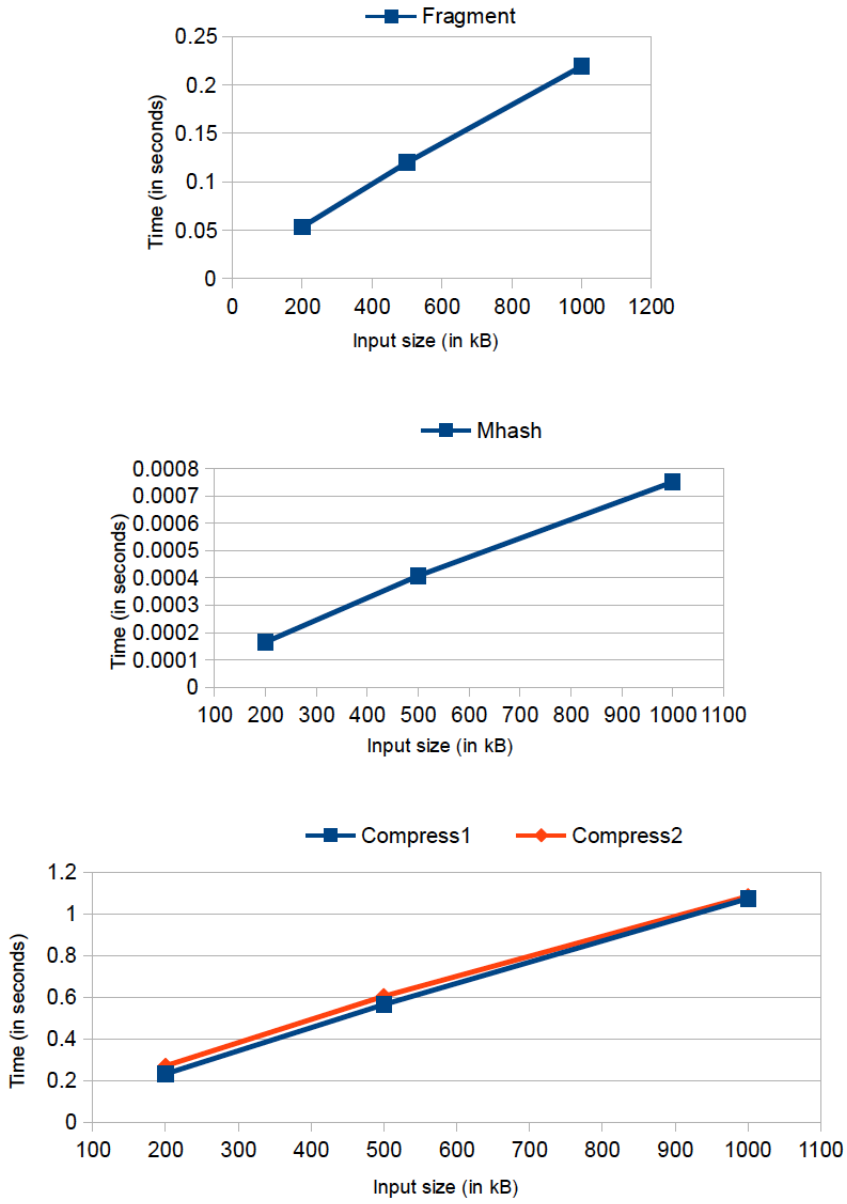


Figure 4.5: Average computation time by process as function of input size.

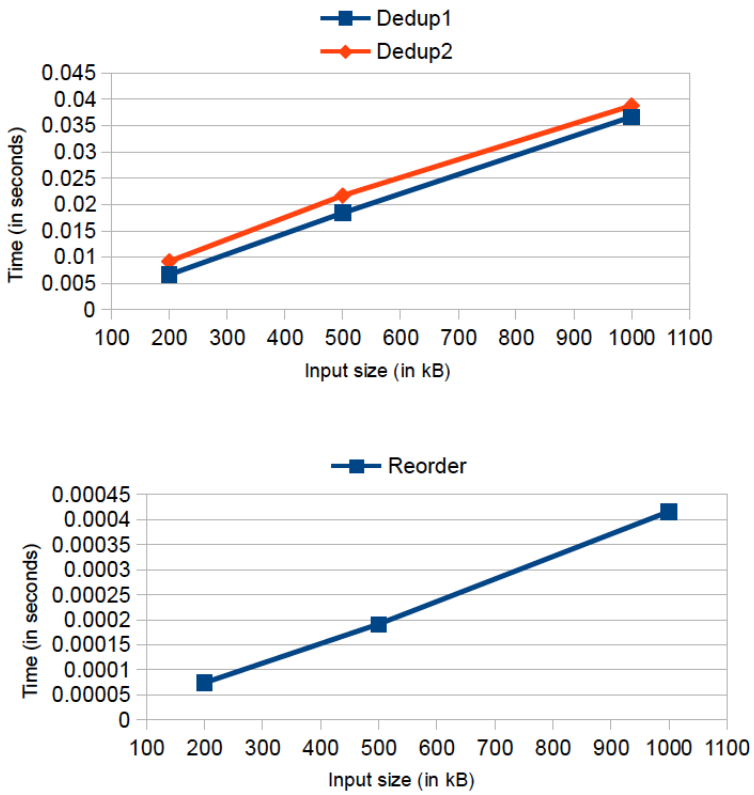


Figure 4.6: Average computation time by process as function of input size.

We can see on fig 4.5 and fig 4.6 that once again timing evolve in a very linear fashion with the input size. What changes significantly however is the ratio of time usage spent for each process. The compression stage is even more prominent than before and the `Mhash` and `Reorder` stages take almost negligible time to complete. The fragmentation process performance can be improved by increasing the coarse chunk size but its performances will alway be limited by the time it needs to load data from memory.

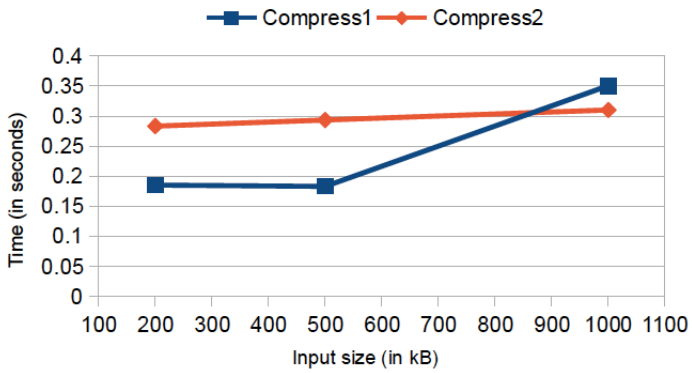


Figure 4.7: Average computation time for compression with duplicated input.

When using a duplicated input, the results are very similar to those of the random input. The only process that behaves differently is the compression process as shown in fig 4.7. Computation time does not increase linearly. Its increase depends on the amount of new data that is discovered since for duplicated data the compression stage is skipped. It can also decrease for one branch depending on what branch of the application discovers the data first.

5 Conclusion and Outlook

5.1 Conclusion

This semester thesis proposed a comparative study of existing parallel benchmarks, the development of a *deduplication* application for many-core architectures in the context of the DOL framework as well as the result of a few benchmarking runs of the application on a PC cluster and an Intel SCC.

This study has shown that the *deduplication* application is a viable choice for benchmarking a many-core architecture because it can be strongly parallelized in a non homogenous fashion (6 different types of process) and can support very big input load. The benchmarking runs are also a proof that this application can be used to benchmark very diverse architectures, from a PC cluster to a Single-chip Cloud Computer.

5.2 Outlook

There are several possibilities to improve and extend the current implementation. First of all the question of the scalability of the application to a very big number of cores (more than 20-30) still needs to be fully addressed. A few improvement could be added to ensure the possibility to work with more processes. If the `Mhash` process were to saturate, it could be split in a few other `Mhash` that would process chunks depending on a few bits of their SHA signature. The `Reorder` and `Fragment` to saturate, the size of coarse

chunks could be increased. It would also be possible to process separate input files at the same time using the same core application and `Reorder` them separately.

Moreover other metrics could be monitored by the application and a comparative study of its performance depending on mapping onto the Intel SCC would provide additional insight on the subject.

A

List of Acronyms

API	Application Programming Interface
DOL	Distributed Operation Layer
FIFO	First-In First-Out
Intel SCC	Intel Single-chip Cloud Computer
I/O	Input/Output
KPN	Kahn Process Network
MPB	Message Passsing Buffer
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration
OS	Operating System
PAPI	Performance Application Programming Interface
PC	Personal Computer
SHA	Secure Hash Algorithm
XML	Extensible Markup Language

B

Presentation Slides

Designing Benchmarks for High Performance Systems

Pierre Ferry

Advisors: Devendra Rai, Lars Schor

Computer Engineering Group TEC
Prof. L.Thiele

Semester Thesis

February 6th, 2013

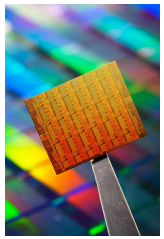
Introduction

1. Introduction
   - Problem description
   - Contribution
2. Benchmark
3. Benchmark results

Problem description

How to compare the performance of multi/many core systems ?

- Very different architectures
- Many performance parameters exist



Contribution

- Comparative study of existing parallel benchmarks suited for high performance processing
- Splitting of each application into multiple tasks
- Porting to Distributed Application Layer (DAL)
- For SCC: Baremetal Specific Design of Communication Layer
- Benchmarks on PC cluster and Intel SCC

Benchmark

1. Introduction
2. Benchmark
   - Benchmark selection
   - SHA application
   - Deduplication application
   - Performance measurements
3. Benchmark results

Benchmark selection criteria

Criteria

- Parallelizability
- Possibility to scale computation and communication loads accordingly to needs
- Realistic application
- Non trivial amount of communication/computation

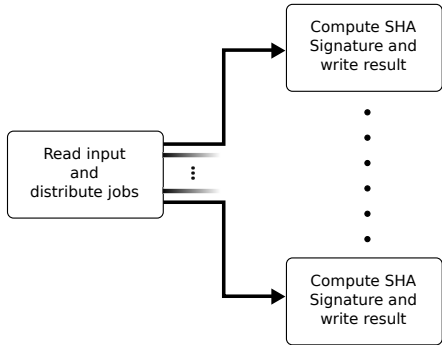
Candidate benchmarks

Benchmark	Parallelism	Complexity	Remark
NASbench	Thread parallelism	Quite complex	Computational fluid mechanics
H264	Many different strategies	Very complex	Video encoding/decoding
ParmiBench	Master/Slave strategies	Relatively simple	Various applications
PARSEC	Diverse strategies	Simple to very complex	Various applications

Secure Hash Algorithm (SHA) application

Secure Hash Algorithm (SHA) application:

- computes the SHA signatures of input files
- simple master/slave strategy



Deduplication motivation

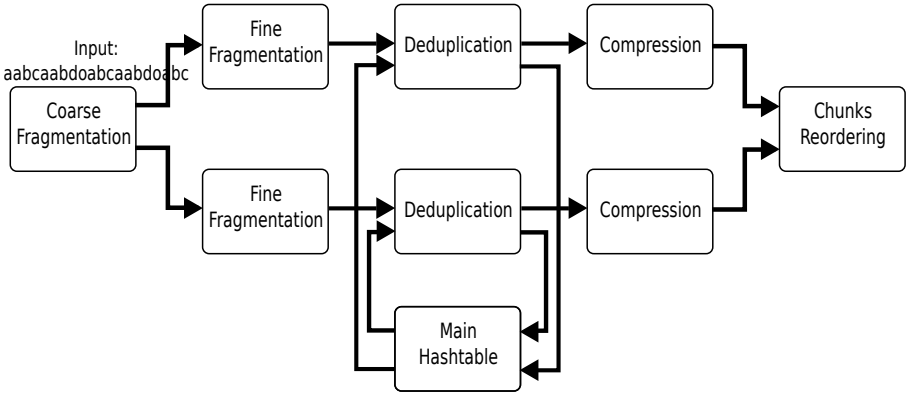
Deduplication is becoming mainstream in data storage :

- Ever increasing data storage needs
- Back-up deduplication
- Inline deduplication

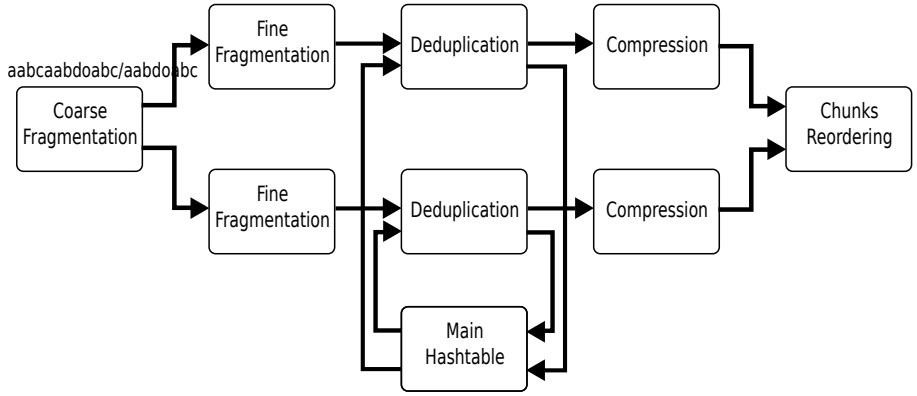
Deduplication is an excellent benchmark because :

- It is a data driven algorithm
- Load scales very well with the input size
- It does non trivial communication/computation
- It is realistic

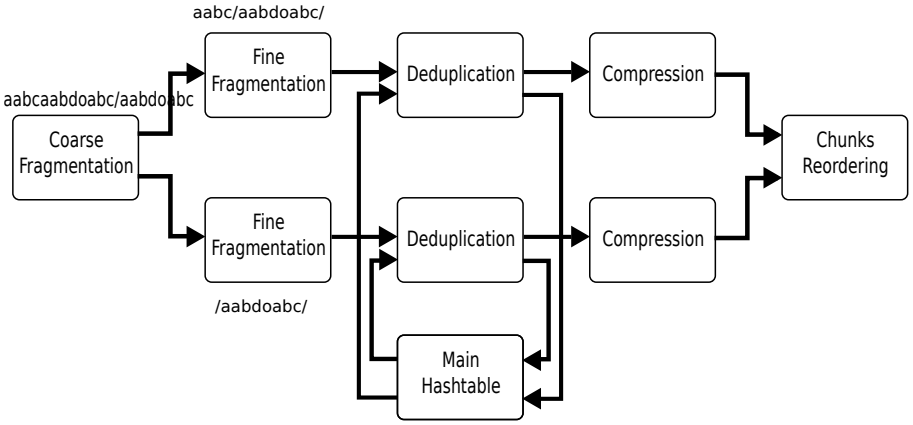
Deduplication method



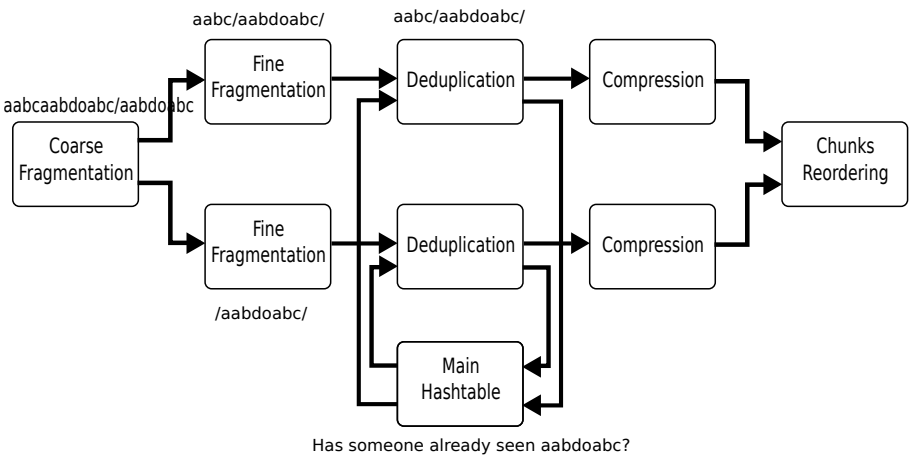
Deduplication method



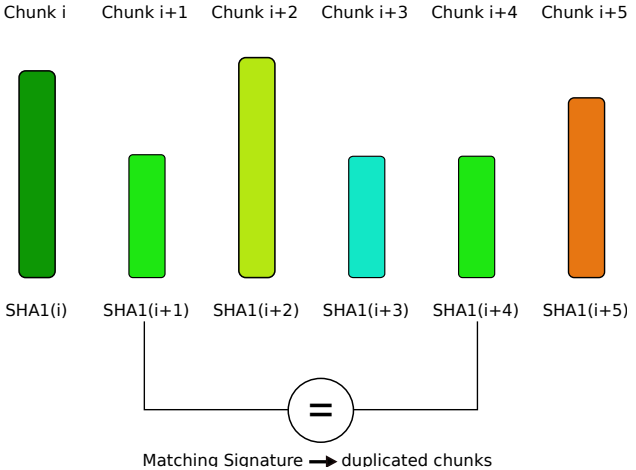
Deduplication method



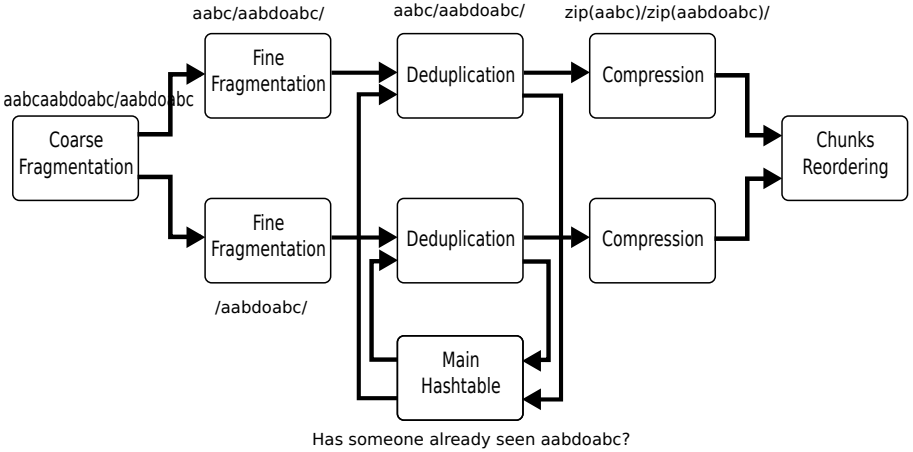
Deduplication method



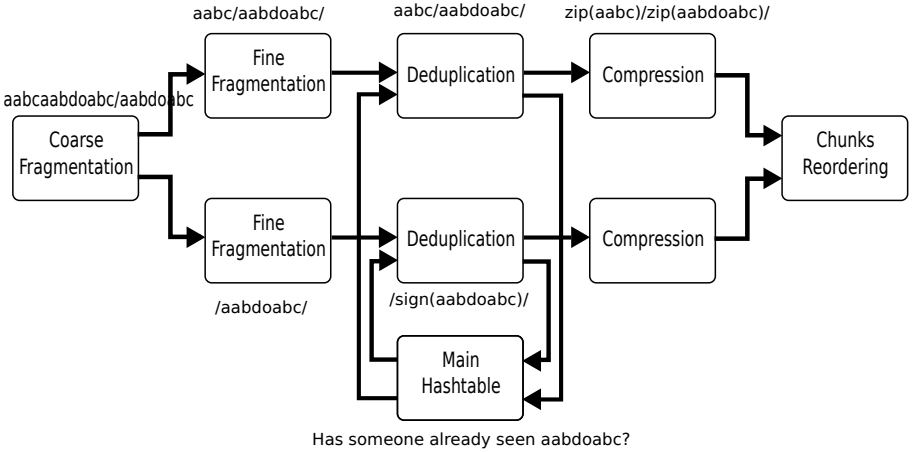
Deduplication method



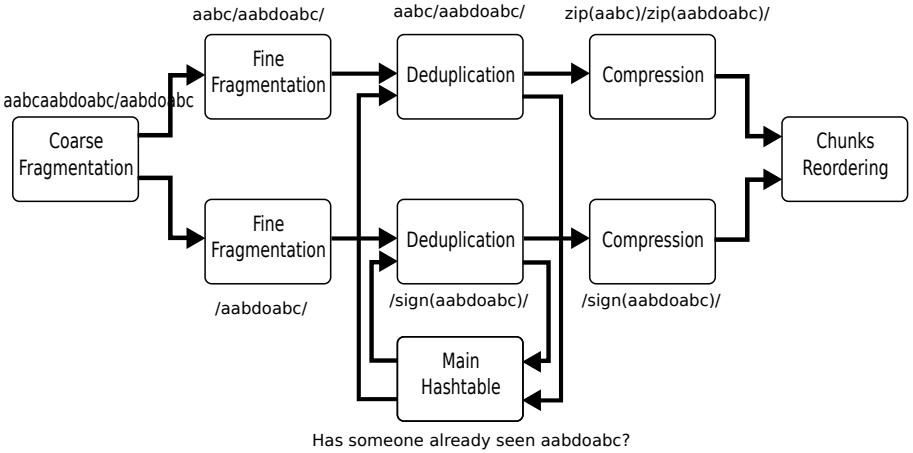
Deduplication method



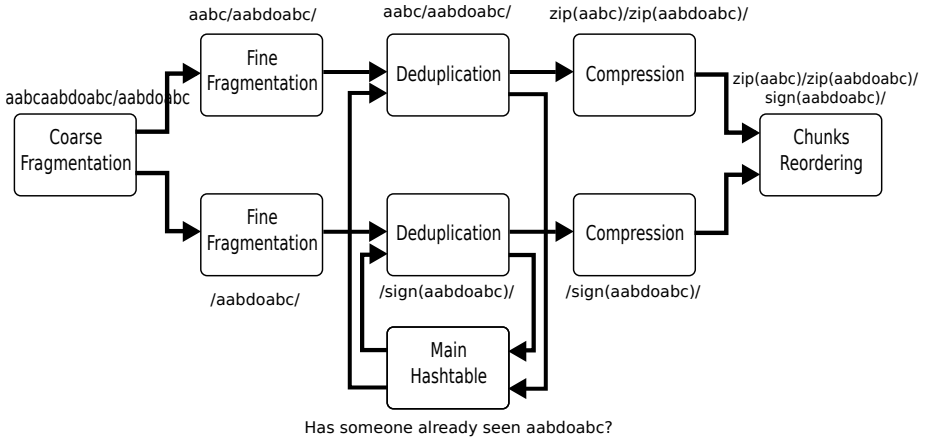
Deduplication method



Deduplication method



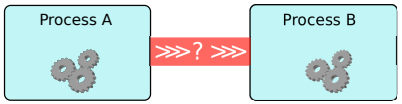
Deduplication method



Measured metrics

This work has focused on:

- Timings
- Hardware cycles
- Communication load

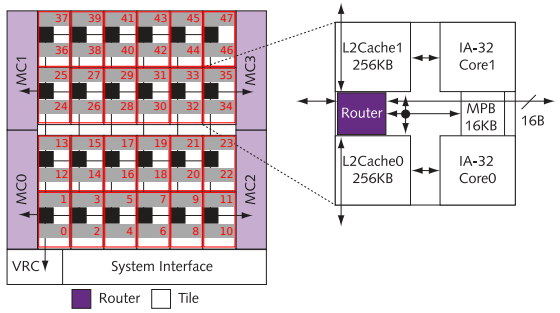


Intel SCC presentation

SCC Layout

Intel SCC characteristics

- 48 cores on chip
- 2 cores per tile at 800 MHz
- Baremetal: no OS
- Accurate timing : 1 process per tile



PAPI library

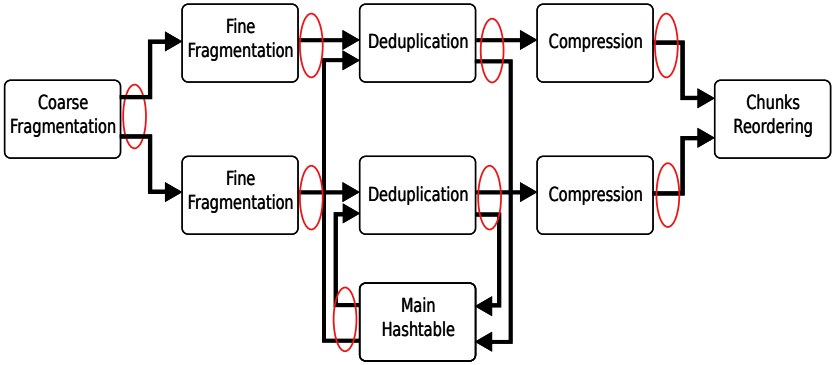
The PAPI library allows us to access hardware counters easily.

- Developed by the Innovative Computing Laboratory from the University of Tennessee.
- Portable, only the lower layers of the API are hardware dependent
- Allows to benchmark a non-homogeneous architecture

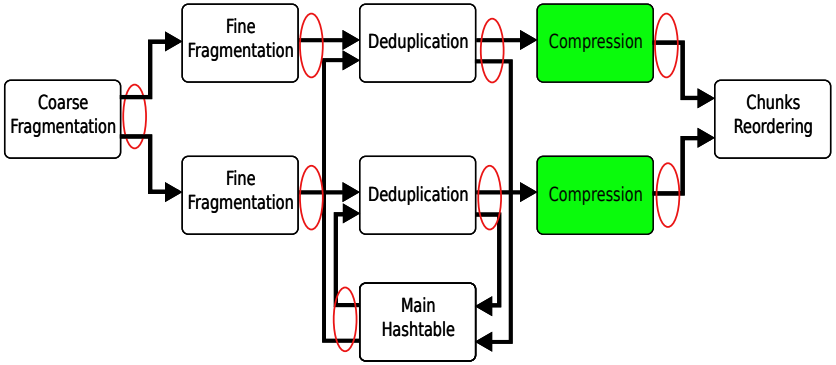
Outline of Topics

1. Introduction
2. Benchmark
3. Benchmark results
   - Communication benchmark results
   - Computation time results

Communication load (1/5)

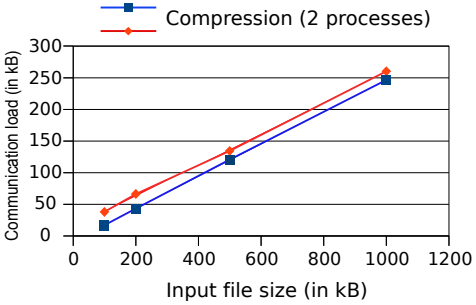


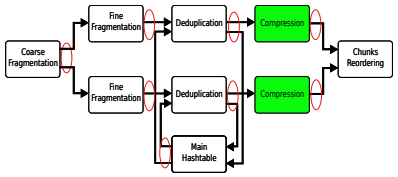
Communication load (2/5)



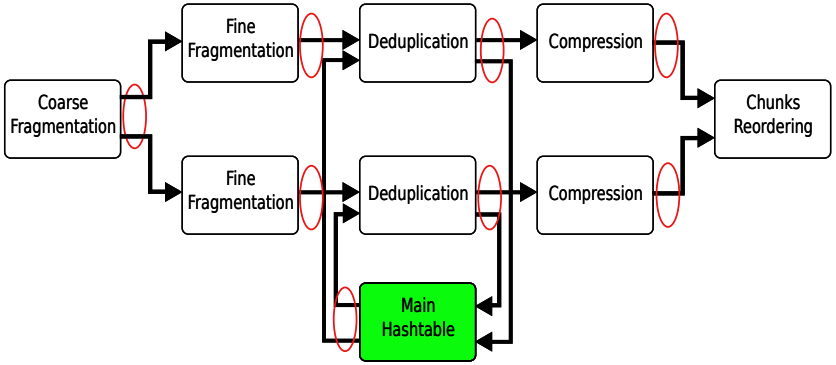
Communication load (3/5)

The communication load of every process increases linearly with the input file size:



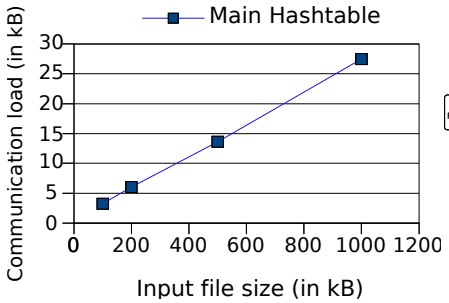


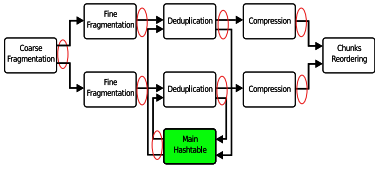
Communication load (4/5)



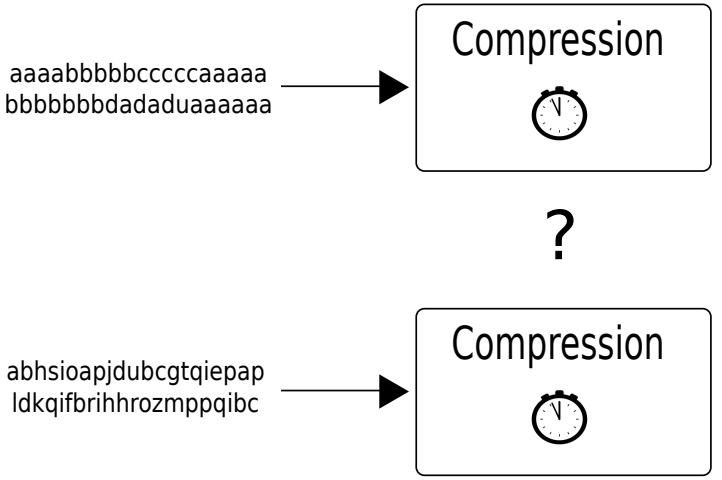
Communication load (5/5)

The Mhash process communicates much smaller amounts of data:



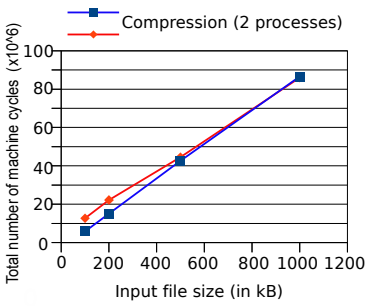


Compression computation time

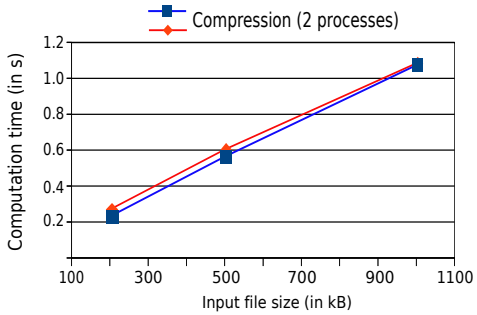


Compression results (1/2)

PC cluster: Machine cycles

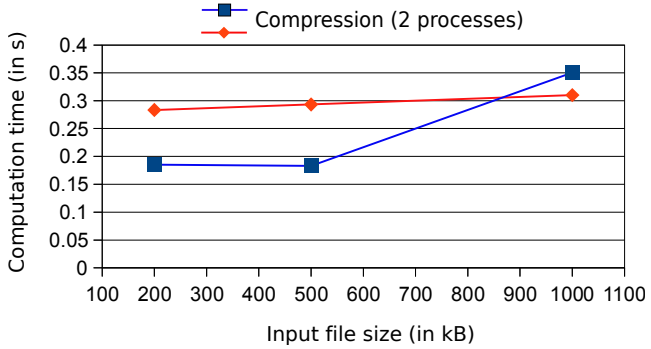


Intel SCC: Timing

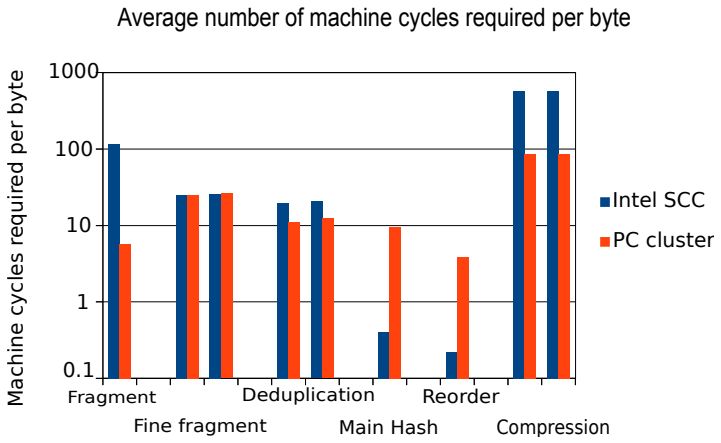


Compression results (2/2)

For duplicated data segments, compression is skipped, hence the computation time does not increase linearly:



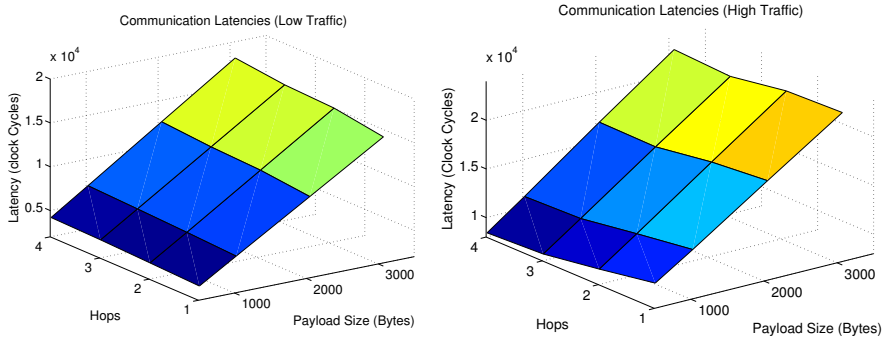
Comparison between computation time of processes



Conclusion

- Comparative study: deduplication
- Porting to DAL and Intel SCC
- Linear scaling of load with input size
- Future development: improve scalability for very high number of cores

Communication latencies on SCC



Bibliography

[1] G. Kahn, "The semantics of a simple language for parallel programming," 1974.

[2] E. L. W. Gropp and A. Skjellum, Using MPI: Portable Parallel Programming with the Message Passing Interface, 1999.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber et al., "The nas parallel benchmarks summary and preliminary results," in Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on. IEEE, 1991, pp. 158–165.

[4] "Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding," 2012.

[5] A. Rodriguez, A. Gonzalez, and M. Malumbres, "Hierarchical parallelization of an h. 264/avc video encoder," in Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on. IEEE, 2006, pp. 363–368.

[6] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer, "Efficient parallelization of h. 264 decoding with macro block level scheduling," in Multimedia and Expo, 2007 IEEE International Conference on. IEEE, 2007, pp. 1874–1877.

[7] A. Azevedo, C. Meenderinck, B. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, and A. Ramirez, "Parallel h. 264 decoding on an embedded multicore processor," High Performance Embedded Architectures and Compilers, pp. 404–418, 2009.

[8] S. Iqbal, Y. Liang, and H. Grahn, "Parmibench-an open-source benchmark for embedded multiprocessor systems," Computer Architecture Letters, vol. 9, no. 2, pp. 45–48, 2010.

[9] C. Bienia, S. Kumar, J. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in Proceedings

of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008, pp. 72–81.

[10] F. Black and M. Scholes, "The pricing of options and corporate liabilities," The journal of political economy, pp. 637–654, 1973.

[11] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in Proceedings of the 6th USENIX Conference on File and Storage Technologies, vol. 18, 2008.

[12] M. O. Rabin, Fingerprinting by Random Polynomials. Center for Research in Computing Technology, Harvard University, 1981.

[13] M. S. Mucci P. and S. N., "Performance Tuning Using Hardware Counter Data," 2001.

[14] GNU, "Ressource Usage," http://www.gnu.org/software/libc/manual/html_node/Resource-Usage.html, [Online].

[15] T. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl et al., "The 48-core scc processor: the programmer's view," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010, pp. 1–11.