EFFICIENT EXECUTION OF SEQUENTIAL INSTRUCTION STREAMS BY PHYSICAL MACHINES

PhD candidate: Ing. Milani Emanuele

Supervisor: Prof. Bilardi Gianfranco

Scuola di Dottorato in Scienza e Tecnologia dell'Informazione

Ciclo XXV

Sommario

Qualsiasi modello computazionale basato su un sistema fisico, è verosimilmente soggetto al fatto che densità e velocità di propagazione dell'informazione sono intrinsecamente limitati. Per questo motivo, il modello *RAM*, in particolare per il presupposto che il costo di un accesso in memoria sia indipendente dalla taglia della stessa, non è implementabile su sistemi fisici.

Questo lavoro si inserisce nel contesto delle *limiting technology machine*, modelli computazionali in cui si ipotizza provocatoriamente di aver raggiunto con la tecnologia di fabbricazione i limiti fisici di densità e velocità dell'informazione. Questo, allo scopo di affrontare il problema delle latenze intrinseche a ogni sistema fisico evidenziando organizzazioni scalabili per processori e memorie.

Viene quindi presentato uno studio algoritmico, che illustra l'implementazione di programmi a elevata concorrenza per SP ed SPE, modelli di macchine sequenziali in grado di eseguire programmi direct-flow in tempo ottimale.

Successivamente, viene introdotta una innovativa organizzazione di memoria, gerarchica e pipelined, con latenza e banda ottimali per un sistema fisico.

Allo scopo di sfruttarne appieno le caratteristiche, e trar vantaggio dall'eventuale instruction level parallelism del codice da eseguire, viene sviluppato un innovativo modello di processore. Particolare attenzione è rivolta all'implementazione di un efficiente flusso di informazione all'interno del processore stesso.

Entrambe le organizzazioni sono estremamente scalabili, in quanto basate su un insieme di nodi a taglia e capacita fisse, connessi con una topologia ad array multidimensionale.

Lo studio delle prestazioni computazionali della macchina risultante ha evidenziato come le latenze interne al processore possono diventare la principale componente della complessita temporale per l'esecuzione di un flusso di istruzioni, che va ad aggiungersi all'effetto dell'interazione tra processore e memoria. Viene pertanto sviluppata una caratterizzazione dei flussi di istruzioni, basata sulla topologia indotta dalle dipendenze tra istruzioni.

Abstract

Any computational model which relies on a physical system is likely to be subject to the fact that information density and speed have intrinsic, ultimate limits. The *RAM* model, and in particular the underlying assumption that memory accesses can be carried out in time independent from memory size itself, is not physically implementable.

This work has developed in the field of *limiting technology machines*, in which it is somewhat provocatively assumed that technology has achieved the physical limits. The ultimate goal for this is to tackle the problem of the intrinsic latencies of physical systems by encouraging scalable organizations for processors and memories.

An algorithmic study is presented, which depicts the implementation of high concurrency programs for *SP* and *SPE*, sequential machine models able to compute *direct-flow* programs in optimal time.

Then, a novel pieplined, hierarchical memory organization is presented, with optimal latency and bandwidth for a physical system.

In order to both take full advantage of the memory capabilities and exploit the available *instruction level parallelism* of the code to be executed, a novel processor model is developed. Particular care is put in devising an efficient information flow within the processor itself.

Both designs are extremely scalable, as they are based on fixed capacity and fixed size nodes, which are connected as a multidimensional array.

Performance analysis on the resulting machine design has led to the discovery that latencies internal to the processor can be the dominating source of complexity in instruction flow execution, which adds to the effects of processor-memory interaction. A characterization of instruction flows is then developed, which is based on the topology induced by instruction dependences.

Contents

to all the people that have been close to me
and in particular to F.M.

Chapter 1

Introduction

The RAM is a model of computation which has allowed much of all the design and analysis of sequential algorithms. Its success is directly related to the essentiality of its definition, combined with the ability to provide for long time fairly accurate performance estimates for large classes of problems.

However, one particular assumption makes the RAM model infeasible in practice. Indeed, time complexity is in general computed by counting the number of basic arithmetical-logic steps necessary to perform the analysed computation, the key assumption being that each state transition, and in particular each memory access, can be performed in constant time, independently from memory and machine size. The likely impossibility derives from two fundamental principles, postulated in [5], which state that both information density and speed have upper bounds in physical systems. The principle of maximum information density, in particular, implies that any system able to store M bits in a d dimensional space, must occupy a volume whose linear dimension, or radius, grows with $\Omega(M^{1/d})$. Due to the principle of maximum information speed, such length results in a proportional latency. When asymptotically evaluating any phisical implementation of the RAM with M bits of memory, it is therefore necessary to take into account a worst case latency of up to $O(M^{1/d})$ time steps for each single operation.

Even if modern machines are still far from being affected by these fundamental constraints, other factors come into play, which results in similar effects. Counting the number of basic arithmetical-logic steps was a very effective way of predicting performance when computing machines were constrained by the switching speed of their circuits. It is more and more inadequate nowadays, as the performance limits are in the form of relatively poor communication latencies and bandwidths among the different components of a computing machine. Indeed, a relevant fraction of the work of designing a new computing machine is already devoted at dealing with these issues.

In the context of sequential computation, two main strategies exist to reduce

the impact of latencies on the overall performance of program execution. First, it is possible to organize the memory as a *hierarchy* of modules with different trade-offs between size and latency, from smaller and faster to larger and slower. If during program execution a relevant part of memory accesses targets the faster modules, it is possible to partially, or even completely in certain cases, hide the cost of latency with respect to the total execution complexity. Programs whose memory accesses are organized in this way are said to exhibit *temporal locality*.

A second strategy consists instead in devising a memory organization which allows *concurrency* of memory accesses. Then, if programs are able to issue memory accesses which can overlap in time, the effect of latencies can again be at least reduced.

Other solutions have been proposed, which offer a combination of both. Models with a *block-transfer* primitive, for example, allow the concurrent repositioning of a *block* of consecutive memory locations across the memory hierarchy. Programs which need to access ranges of consecutive memory locations derive particular benefits from such an organization, and are said to exhibit *spatial locality*.

The seminal work on memory hierarchies and program locality lead to the definition of models such as the *External Memory Model* [17] and the *Hierarchical Memory Machine* [1]. The *BT-RAM*, or RAM with block transfer, was introduced in [2], the relevance of the model deriving also from the fact that a lot of actual hardware currently relies on such operation. The *Logarithmic Pipelined Model* [14] on the other hand, explores non-hierarchical memories with pipelined access, and analyzes algorithmic complexity based on their concurrency.

A more fundamental approach is taken instead in [4], which deeply investigates possible realizations of the abstract RAM taking into account the fundamental limits on information density and speed. In this spirit, such realizations are designs which specify a layout in d dimensions, so that the time for each state transition is independent from machine size. In other words, the designs are scalable to arbitrary machine size, as advocated in [5].

The resulting models feature a memory organization which is both hierarchical and pipelined, so to be able to accept one request per time step, or at a *constant rate*, and bound the latency to the access time of the queried memory module. Dedicated processor organizations, the *SP* (*Speculative Prefetcher*) and the *SPE* (*Speculative Prefetcher and Evaluator*), are adopted in order both to take the most advantage from a pipelined hierarchical memory, and to avoid unnecessary accesses by managing information flow within the processor itself.

The work which is presented here stems from the results of [4] in two ways. The first part is a further algorithmic exploration of SP and SPE. In particular, it is shown how parallel algorithms designed for the *Work-Time Framework* [11] can be encoded into programs with a high concurrency of memory accesses. While the

fundamental idea is not new (see for example [14]), its realization in the context of SP and SPE is not straightforward, and leads to some new insights into the two models. The results have already been published in [15].

The second and main part of this work, instead, extends the investigation of scalable machine designs to explicitly parallel machines. Indeed, while [4] restricts its scope to sequential computation, such constraint is by no means dictated by fundamental limits. On the contrary, the analysis in [5] and [6] show that parallel organizations can be extremely effective choices for scalable machines.

It is interesting to note that a shift towards parallel architectures is an important and pervasive trend also in actual machines. Reasons for this phenomenon include, among the others, the increasing difficulty of improving single core performance, the possibility of implementing innovative forms of power management, the emerging of new kinds of workloads, yield improvements.

A novel machine organization, with separate processor and memory subsystems, is therefore presented.

First, an organization for a high-bandwidth, pipelined, hierarchical memory is introduced. Its design specifies a layout in d dimensions for a memory of M locations, which offers a bandwidth of $O(M^{(d-1)/d})$ at a sustained rate, with a latency depending only on the levels which implement the requested locations.

Then, a specialized processor is presented, in order to both fully exploit the capabilities of the memory system, and enable concurrent execution of instructions.

Much of the effort for specifying the processor goes to an efficient management of the internal information flow, so that at least some of the instruction communications which traditionally involve multiple accesses to memory, can take place within the processor itself.

Both memory and processor are designed in a extremely modular way. Larger machines can be implemented with the simple addition of more modules.

One insteresting result in the analysis of the capabilities of the new model is that the latencies due to communication internal to the processor can be the dominating figure in the overall execution complexity, which adds to the effects of processor-memory interaction.

A characterization of instruction flows is then developed, which is based on the topology induced by instruction dependences. It is shown in particular, that the ratio between the length of the longest dependence chain and the number of instructions is a good metric to capture the behavior of code with respect to internal processor communication.

The aim of the proposed machine design is the efficient execution of *sequential* programs. There are multiple reasons for this. First of all, sequential algorithms and code constitute a relevant part of the available background in informatics and engineering. Then, parallel programming is still struggling to become the main

programming paradigm. Indeed, besides being inherently more complex to design, analyse and debug parallel code or algorithms, there in no dominant framework, or model, which is able to both express machine capabilities and be flexible enough with respect to the users' needs. On the contrary, the shift from sequential towards parallel programming would be best described by a fragmentation rather than a unitary trend.

On top of that, also in the context of parallel programs, the ultimate constraint to the achievement of execution concurrency are sequential portions of code [3]. It is therefore of paramount importance that a general purpose parallel machine is able to achieve the best possible performance with sequential code.

Next is an overview of this work. Chapter 1, besides this introduction, provides some background information on instruction dependences and the concept of scalable machines.

In Chapter 2, the algorithmic results on SP and SPE are presented, highlighting the different approaches needed with the two different processors. Applications of the findings lead in particular to an optimal linear-time implementation for SP and SPE of a program for merging two sorted lists.

Parallel machines are introduced in Chapter 3. A high-bandwidth non-hierarchical, pipelined memory is initially described, to be then used as a module in a hierarchical memory organization. Extra care is devoted to the geometrical placement of memory modules, and the relative position between them and the processor, in order to preserve the latencies and bandwidth offered by the single modules.

The processor organization is later introduced, starting from the definition of its requirements and proceeding with the description of its operating behavior. An important concept is the distinction between static programs and dynamic instruction stream, which are the sequence of instructions which are actually computed on a given input.

Chapter 4 investigates lower bounds which affect sequential execution complexity on two-dimensional arrays. The results show how the topology induced by instruction dependences can be the dominant component of execution complexity, overgrowing the impacts of memory latency and bandwidth.

Chapter 5, then, focuses on the strategies which implement execution itself. After presenting a general purpose, but not so efficient strategy, it is shown how extra information, in the form of instructions annotations, can dramatically improve the time complexity. To this purpose, three strategies of increasing complexity and efficiency are introduced, the last of which is able to match some of the lower bounds of Chapter 4.

Chapter 6, finally, provides an overview of the achieved results and depicts the possible paths that could be taken to proceed the investigation on physically implementable machines.

1.1 Programs and Dynamic Instruction Streams.

In this Section, some useful concept are reported from [4], as they provide the basis for most of the presented results.

A sequential program is in its essence a list of instructions. In turn, each instruction specifies an operation and a list of operands which are functional to the operation itself.

Operands include a value and an addressing mode, which can be *literal*, if the operand represents the value itself, *direct*, if the value is a memory reference, or *indirect* when the value is a reference to a reference.

During program execution, when the same locations are accessed more than once, the order of accesses induces dependence relations among the originating instructions.

At this point, it is useful to introduce a further distinction among sequences of instructions. While the list of instructions which constitute a program is fixed, the sequence of instructions which are actually executed by a sequential processor is not, and in general depends on the evolution of memory states caused by execution.

Definition 1. A *dynamic instruction stream* is the stream of instructions which are executed by a sequential processor which is given a program and an initial memory state.

The dependences among instruction which are targeted in this work can be classified in two categories.

Definition 2. There is a *functional dependence* between instructions i and instruction j when the output of i is used by j as an operand.

Definition 3. There is an *address dependence* between instructions i and instruction j when the output of i is used by j as the address of an operand.

The simplest approach for dealing with dependences is letting all the necessary communication happen through memory accesses. As it can be expected, there are better choices in terms of time complexity. Indeed, it is in general possible to avoid paying some of the latencies deriving from these accesses.

It is important to point out that address dependences have a much heavier impact on execution complexity. Indeed, the entire communication between two instruction involved in a functional dependence can take place in the processor itself. This is not the case, instead, of address dependences, for which a memory lookup has to take place between the executions of i and j.

The operations specified by the instructions can be of *data processing* or *control flow* type. While the former is typically an arithmetic logic operation, the effect of the latter is on the selection of the next instructions to be executed.

Two categories of programs are of particular interest.

Definition 4. A program is *straight line* if it is composed only of data processing instructions.

Definition 5. A program is *direct flow* if it is straight line and the operands specified by its instructions have only direct or literal addressing mode.

1.2 Limiting Technology Machines

An interesting approach in the investigation of the concept of scalability in the context of limited information speed and density is the one given in [5]. The somewhat provocative scenario which is described in this work, is that of *limiting technology machines*, which are computing machine whose circuits implementation matches the computational limits of physical systems. In other words, it is not possible to achieve further performance gains through furter miniaturization or operating frequency increase.

The purpose of this theoretical exercise is that of devising machine designs which are optimal independently from the adopted produciton technology.

In these setting, the only way to build machines with increase performance is to assemble larger machines. The main difficulty arises from the necessity of decoupling the time complexity of a state switch from the size of the machine itself. The approach which is taken in this work is the same as in [4], and consists in structuring machine designs as a bounded degree interconnection of nodes of fixed capabilities and size. In this way, state transitions only involve a fixed amount of logic ports and space, thus making it possible to be carried out in time independent from machine size. The concept is further resumed with the presentation of the parallel design in Chapter 3.

Chapter 2

Sequential Limiting Technology Machines

This Chapter presents a technique to encode a parallel work time-framework algorithm to a sequential program suitable for execution on the SP or SPE.

SP and SPE are two processor designs introduced in [4] in order to be able to fully exploit the capabilities of a pipelined hierarchical memory.

One essential feature is that instructions are processed in segments, and not one by one. Memory accesses of instructions belonging to the same segment are aggregated, so to overlap them and exploit the concurrent accesses to memory. Besides, the result of each instruction execution is propagated to the following ones, thus eliminating the need fo some of the memory access.

Clearly, the generation of the segment of instructions to be executed is not in general a deterministic process. Indeed, whenever a control-flow instruction is met, it becomes necessary to predict the result of the branch, for later checking its correctness.

Both designs are scalable to arbitrary size, in that they are implemented as linear arrays of nodes of fixed size. Therefore, the time for a state transition is decoupled from the size of the machine.

After reviewing some important details about the SPE and introducing an essential background on PRAMs and the work-time framework, a technique is presented to implement high concurrency programs for the SP and SPE. Then, an improved version, with a better memory utilitation is presented, which only runs on the SPE.

Finally, after showing the conditions for optimality, examples which address practical problems are considered.

2.1 The Speculative Prefetcher and Evaluator

The Speculative Prefetcher and Evaluator is a processor design which is intended to fully exploit a *Pipelined Hierarchical Memory* (PHM) while complying with the physical constraints of maximum information density and speed (see [5]). Both memory and processor are extensively discussed in [4].

The PHM includes M memory locations, each of which is addressable by means of a unique integer in $[1, M]$. Its hierarchical nature is captured by a *memory access function* $a(x)$, which expresses the latency of a request for memory location x. Without loss of generality, locations are numbered such that $a(\cdot)$ is a non-decreasing function. Besides, by its pipelined nature, the PHM is able to accept one request per clock cycle; in other words, it is able to sustain a *constant access rate*. As a consequence, it is possible to hide latencies by overlapping. In fact, the total latency of q independent requests is $q + a(M_{max})$, M_{max} being the slowest accessed location. On the other hand, any pipelined not hierarchical memory always accounts for a $a(M)$ latency, thus making it more and more difficult to amortize it when M grows. Moreover, it should be noted that, for any pure hierarchical memory, as the amount of locations to be accessed grows, access rate asintotically drops to zero.

As shown in Figure 2.1, the SPE includes an *Instruction Generator Unit* (IGU), connected to an *Instruction PHM*, and an *Instruction Execution Unit* (IEU), connected to a *Data PHM*. While actual instruction execution is carried out by the IEU, the task of the IGU is to translate a static program into a dynamic instruction stream. Both IEU and IGU are linear arrays of k constant-sized units called *stations*, each of which can store and manage one instruction. Parameter k, hence, denotes processor size.

Execution consists of a sequence of *stages*. At each stage the IGU fetches instructions from the Instruction PHM and assembles a *segment* of $q \leq k$ dynamically loaded instructions to be supplied to the IEU. In doing so, the IGU has the ability to perform branch predictions, and consequently loop unrolling. Segment length q can be programmatically adjusted within the $[1; k]$ range via the `segmentsize()` directive.

The IEU executes the segment in a series of *rounds*. Each round is divided into two parts: in the first the IEU speculatively fetches the operands, both directly and indirectly addressed, of all the instructions of the segment; in the second, all the stations compute sequentially with the speculative operands they hold. In case speculation is successful, the SPE can effectively hide memory access latencies. Anyway, speculation may fail due to dependences among instructions. In particular, two kinds of dependences may arise, which are dealt with in different ways.

In the following definitions, $m[x]$ indicates the content of memory location x.

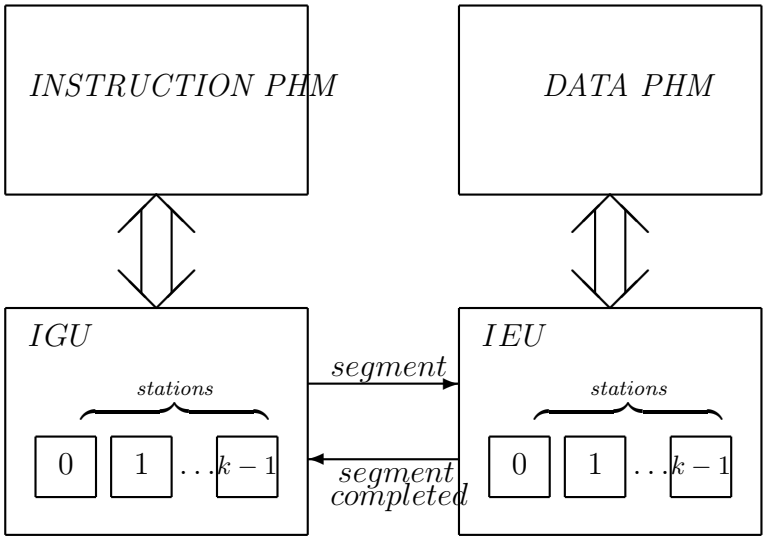

Figure 2.1: Scheme of the PHMM architecture: IGU reads instructions from the *instruction PHM*, produces a *segment* and passes it to IEU stations. Here data are speculatively prefetched and instructions are executed, solving possible invalid data with the internal forwarding or reading them from *data PHM*.

Definition 6. There is *Functional Dependence* (FD) between instructions I_j and I_k if I_j modifies $m[x]$, and I_k uses the content of $m[x]$ as operand or to indirectly address the output location, while no operation between I_j and I_k modifies $m[x]$.

Definition 7. There is *Address Dependence* (AD) between instructions I_j and I_k if I_j modifies $m[x]$, and I_k uses $m[x]$ to indirectly access an operand, while no operation between I_j and I_k modifies $m[x]$.

Functional dependences can be effectively dealt with via a mechanism of *internal forwarding*. In the IEU, whenever the result of an instruction is ready, beside being committed to memory, it is propagated to the following stations. This way, in case of a functional dependences, the updated values are available in the IEU without further memory accesses.

On the other hand, an address dependence requires that a memory access takes place between the computation of the dependent instructions. Its latency, therefore, can be hidden only if enough instructions are interposed between the dependent ones.

The time complexity of a round (initialization and execution) is $O(q+a(M_{max}))$, q being the size of the segment and M_{max} the slowest accessed memory location. By carefully balancing q and $a(M_{max})$ it is possible to jointly exploit locality and concurrency of accesses, and hide latencies. For the same reason, a sensible choice for processor size k is to match the worst case latency $a(M)$.

When speculation is effective, only few rounds suffices for completing a segment, and it is possible to achieve $O(1)$ amortized complexity per instruction.

In order to understand the impact of dependences on complexity, we recall the definition of some concepts.

Definition 8. Given the instruction stream $(I_1, I_2, \ldots, I_N)$ generated by the IGU during the execution of program $\mathcal{P}$ on a particular input, its *address dependence depth* D is the maximum length of a subsequence $I_{j_1}, I_{j_2}, \ldots, I_{j_L}$ with $j_1 < j_2 < \cdots < j_L$ where subsequent instructions have address dependence.

Definition 9. A program $\mathscr{P}$ is *straigth–line* if it consists only of data processing instructions.

Definition 10. A program $\mathscr{P}$ is *direct–flow* if it is straigth–line and does not use indirect addressing.

In [4] it is shown that any N-instruction straigth–line program with address dependence depth D and accessing memory locations with address smaller than M can be executed by an SPE in time $T = O((D+1)(N + a(M)))$. Moreover, for a direct–flow program, $D = 0$; hence, it can be executed in $T = O(N + a(M))$.

In particular the following result from [4] holds:

Lemma 1. *A program consisting of nested for loops where the only branches are those related to the cycles, can be executed in* $T = O(D(N + a(M)))$.

Lemma 1 applies to wide classes of programs, such as FFT and Matrix Multiplication.

We quote the following example, which intuitively shows how this occurs.

Example 1. Let us consider the execution of a C-like code that increments every element of an array: `for i=1 to k; A[i] = A[i]+1`. Using the naive branch prediction policy that always reenters the loop, the IGU can unroll the loop in `i=1; m[i]=m[i]+1; i=i+1; m[i]=m[i]+1; ....` The IEU speculatively calculates all the i values at the first round; then, at the second one, prefetches all the indirect accessed operands, and correctly completes the segment through speculative execution. So it can resolve address dependences in O(1) amortized time.

2.2 PRAMs and the Work-Time Framework

A *Parallel Random Access Machine* (PRAM) [9, 10] is an abstract parallel machine model, that consists in a collection of P synchronous processors and M shared memory locations. A *PRAM program* is a sequence of parallel steps, each of which specifies an instruction per processor.

Beside the number of nodes, the computational power of a PRAM is determined by which shared memory operations are permitted. Within a step, in fact, each memory location may or may not be accessed by more than one processor. In other words, a PRAM can be provided with either an *exclusive read* (resp. *exclusive write*) memory, or a *concurrent read* (resp. *concurrent write*) memory. Moreover, when concurrent writes are allowed, a contention policy must be specified in order to determine the actual memory state after the access. For example the *priority* policy allows only the processor with highest priority to write on the contested location, while the *common* policy allows concurrent writes on a location only if all the involved processors are writing the same data. The most studied configurations, in order of increasing power, are exclusive read exclusive write (EREW), concurrent read exclusive write (CREW), concurrent read concurrent write (CRCW).

Both Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) versions have been studied. Anyway, they are equivalent [8] if they feature the same memory access policy. In this work, we will just refer to the SIMD version.

The *Work-Time framework* (WT) [11] is a parallel programming framework in which an algorithm $\mathscr{A}_{WT}$ consists in an ordered sequence of T sets $s_0, \ldots, s_{T-1}$ of independent operations on M_{WT} memory locations. Different sets may differ in size and therefore exhibit more or less parallelism. Let $|s_i| = p_i$, then we define the *work* W of $\mathscr{A}_{WT}$ as $W = \sum_{i=0}^{T-1} p_i$.

WT algorithms are meant to be executed by PRAMs, by means of a schedule. A sufficient condition for a valid schedule of $\mathscr{A}_{WT}$ in a PRAM is that each operation in s_i is executed after all operations in s_{i-1} and before any operation in s_{i+1}. Any such schedule allows us to apply *Brent's Theorem* [7] and to execute $\mathscr{A}_{WT}$ in a PRAM with P processors in a time $O(\frac{W}{P} + T)$.

On the other hand, it is not clear how we can reschedule a PRAM program for P processors as the processor number increases, since possible dependences between steps are not stated explicitly. For this reason WT framework is much more convenient if we need to extract dependences and available parallelism.

It should be noted that, since it is always possible to simulate $\mathscr{A}_{WT}$ on a RAM in time $T_{RAM} = W$, lower bounds on RAM complexity automatically hold also for the work. In particular, let T^*_{RAM} be the best RAM complexity for a given problem. Then, the equivalent WT algorithm $\mathscr{A}_{WT}$ is *work–optimal* if and only if W is $O(T^*_{RAM})$.

The major construct of the WT model is the `pardo`, which specifies a parallel step with a syntax similar to a traditional `for`. The main difference is that the cycle index denotes just the index of an element of the set of instructions and can not be modified by the instructions. For example

```
for j, 1 ≤ j ≤ p pardo
    operation_j
```

denotes a set of p independent operations, whose execution order is irrelevant.

Since WT framework is a very high level model, it is important to pay attention that:

F1 in each parallel step all the reads take place before any write;

F2 addresses can be expressed in a high level fashion.

Therefore any simulation or implementation of an algorithm which relies on such features has to provide them. We address these issues in two steps: first we show how every algorithm can be reformulated in a less general yet equivalent (from an asymptotic time and space complexity point of view) form, which does not require hypotesis F1; then we show how to derive efficient SPE programs from this special sub-class of WT algorithms.

Let us introduce a class of WT algorithms.

Definition 11. A step of a WT algorithm $\mathscr{A}_{WT}$ is *CRCW decoupled* if any concurrently accessed memory location is either read or written. $\mathscr{A}_{WT}$ is itself *CRCW decoupled* if this condition holds for each step.

Any CRCW decoupled algorithm does not rely on (F1). In the opposite case, it is possible to devise an equivalent CRCW decoupled algorithm with the same work and time complexity. In fact, it suffices to split each parallel step into two sub–steps. The first fills an auxiliary array with the operation inputs, while the second performs the actual execution, reading from the array. In the worst case, the memory overhead is $O(p)$.

Consider now, without loss of generality a SIMD parallel step in the WT framework, which executes on an initial memory state $\mathscr{M}_i$ and leads to final state $\mathscr{M}_{i+1}$. Available parallelism and the memory locations that must be read or written (both usually parametrized with the size of the input) are indicated by a `pardo` statement. In particular an index j is used to distinguish each concurrent operation. Note that the memory to store the instructions for a whole parallel step is constant and therefore the whole program takes $O(T)$ memory.

In particular, the operands of operation j are a function of j. Typically, such function is simple enough to be expressed by the addressing modes of a modern instruction set (for example a base address and an index–dependent offset). In the most general case, when (F2) is fully exploited, the function can be explicitly used to prepare an auxiliary operand vector.

2.3 High-Concurrency SPE Programs

One way to write efficient programs for SPE is to exploit the parallelism of Work–Time algorithms. Parallel steps can be efficiently coded into sequential programs, also in case of concurrent memory accesses, which can be sequentially implemented with little effort. In fact, the address dependence depth of each resulting segment is $O(1)$, and memory accesses can be fully pipelined.

However, it must be noted that a straightforward static unrolling of a `pardo` statement could lead to an SPE program with $O(W(n))$ size. In this case, instruction fetch latencies could be larger that data latencies, thus hindering time efficiency.

A more complicated unrolling, proportional to processor size, can be devised, which leads to programs with $O(k)$ size. This last strategy is not the most compact, still it is interesting because it also applies to the SP processor (see [4]).

We now address the problem of executing a parallel step s_i of p_i independent operations of a Work-Time algorithm in an SPE of size k. The set s contains p instructions a_i,

$$s_i = \{a_i(x_{i,0}, x_{i,1}, x_{i,2}) : i = 0, 1, \ldots, p_i - 1\},$$

where i is the operation id and $x_{i,j}$ is the jth operand of ith operation. We recall that we deal with SIMD programs, therefore every operation of the set s has the same opcode. Operand $x_{i,j}$ can represent the actual operand, the memory address containing the operand (direct addressing) or the memory address for the indirect addressing and it is possibly null (operations have from one to three operands).

Our strategy requires $q = O(\min(k, p))$ extra memory cells, which do not augment the latency since $q = O(\min(k, p)) = O(k) = O(a(M)) = O(M)$. It consists of a loop, which considers q operations of s at once. The loop is divided in three parts

1. generation of the array `IDX` containing the instruction indeces;
2. generation of the array `OP` containing all the operands and the addresses of output cells;
3. execution of the segment as a direct flow.

In the following there is the code for the case $p < k$ or p multiple of k, which can be easily extented to arbitrary p managing the elements of last segment after of the loop.

$q \leftarrow \min\{p, k\}$
`segmentsize(`q`)`

```
for j, 0 ≤ j ≤ ⌊p/k⌋ do {
      // part 1
      IDX[0] ← j · q
      IDX[1] ← j · q + 1
      ...
      IDX[q-1] ← j · q + (q − 1)
      // part 2
      OP[0] ← x_{IDX[0],0}
      OP[1] ← x_{IDX[0],1}
      OP[2] ← addr(x_{IDX[0],2})
      ...
      OP[3(q-1)] ← x_{IDX[q−1],0}
      OP[3(q-1)+1] ← x_{IDX[q−1],1}
      OP[3(q-1)+2] ← addr(x_{IDX[q−1],2})
      // part 3
      instructions_{IDX[0]}(OP[0], OP[1], OP[2])
      instructions_{IDX[1]}(OP[3], OP[4], OP[5])
      ...
      instructions_{IDX[q−1]}(OP[3(q-1)], OP[3(q-1)+1], OP[3(q-1)+2])
}
```

The loop is executed $\lfloor \frac{p}{k} \rfloor + 1$ times, with a total complexity of $O((\lfloor \frac{p}{k} \rfloor) a(M) + a(M)) = O(p + a(M))$. The code size is $O(q) = O(\min(k, p))$.

Theorem 2. *Consider WT algorithm $\mathscr{A}_{WT}$, with $W(n)$ work and $T(n)$ parallel steps. Then an equivalent SPE program $\mathscr{P}_A$ can be written, with $O(W(n) + T(n)a(M_{total}))$ complexity and $O(k)$ program size, where M_{total} is the overall memory used by $\mathscr{A}_{WT}$ and k is the size of the SPE.*

2.4 High-Concurrency, Compact SPE Programs

It is possible to exploit the loop unrolling and speculative execution capabilities of the SPE to obtain a better result than that of Section 2.3. Without loss of generality (see Section 2.2), let us restrict our scope to CRCW decoupled WT algorithms. The resulting method is itself rather simple. If only exclusive writes are used, WT statement

```
for j, 1 ≤ j ≤ p pardo
    operation_j
```

can be coded for SPE with a loop in the form:

```
segmentsize(min(k, p))
for j, 1 ≤ j ≤ p do
    instructions_j
```

where $\texttt{instructions}_j$ is the bare SPE coding for the high level WT $\texttt{operation}_j$.

As for concurrent writes, contention policies are quite different one from another, and therefore different approaches are needed for their implementation. Consider, as an example, a reduction–like policy, based on an associative and commutative operation as MAX, + or logical AND. In the SPE it is sufficient to append the appropriate reduction instruction to the core of the loop. The resulting code has the following structure:

```
segmentsize(min(k, p))
for j, 1 ≤ j ≤ p do
    instructions_j
    acc ← max{acc; output_j}
```

where $output_j$ is the result of $\texttt{instructions}_j$, the reduce operation is a MAX and the final result is accumulated in variable *acc*.

Another common policy, *priority* CW PRAM, can be implemented recurring to *predicated instructions*, whose output is committed to memory only if a certain condition is verified. The following example code refers to a situation where only one location is concurrently written to.

```
currentMin ← p + 1
segmentsize(min(k, p))
for j, 1 ≤ j ≤ p do
    cond ← j <currentMin
    [cond]currentMin ← j
    [cond]instructions_j
```

where `[cond]instruction` commits to memory only if the boolean `cond` is true. `currentMin` is efficiently updated within the processor via internal-forwarding.

It should be noted that with this method the number of lines of the resulting program is proportional to the length of the original algorithm, and therefore independent from input size.

The correctness of the simulation relies on the following facts:

- SPE instructions are chosen to match the corresponding WT operations;
- each SPE instruction receives the right operands;
- memory writes are consistent with the policy specified by $\mathscr{A}_{WT}$.

Hence, Lemma 3 holds true.

Lemma 3. *Given WT parallel step s_i, it is possible to implement an equivalent SPE program $\mathscr{P}^{(i)}$, such that they both lead from memory state $\mathscr{M}_i$ to $\mathscr{M}_{i+1}$.*

Proof. As already mentioned, the body of the for loop in the SPE program correctly implements the operations of the Work-Time algorithm by construction, and in both cases the same operands are used. □

Next, we need to prove that this method actually leads to efficient solutions. Before examining the actual time complexity of the resulting programs, it is necessary to consider and measure their memory consumption. In fact, the latencies that must be paid heavily depend on that.

We start by providing bounds on the amount of memory required by the SPE implementation $\mathscr{P}^{(i)}$ of a single parallel step s_i.

Lemma 4. *Let $M_{WT}^{(i)}$ (resp. $M_{WT}^{(i+1)}$) be the size of memory state $\mathscr{M}_i$ (resp. $\mathscr{M}_{i+1}$), and let $M_{PH}^{(i+1)}$ be the amount of memory needed by $\mathscr{P}^{(i)}$. Then, both $M_{PH}^{(i+1)}$ and $M_{WT}^{(i+1)}$ are $O(M_{WT}^{(i)} + p_i)$.*

Proof. Since p_i is the number of operations of the parallel step, $M_{WT}^{(i+1)}$ is $O(M_{WT}^{(i)} + p_i)$. As for $M_{PH}^{(i+1)}$, an auxiliary operand vector only needs $O(\min\{p_i, k\}) = O(p_i)$ extra space. □

This bound can be extended to the entire SPE program in a direct way.

Lemma 5. *Let $M_{total} = \max_i\{M_{PH}^{(i)}\}$ be the maximum memory needed by any of the $\mathscr{P}^{(i)}$ which implement some step s_i of $\mathscr{A}_{WT}$. Then M_{total} is bounded from below by the maximum available parallelism of $\mathscr{A}_{WT}$ plus input size n and from above by the work $W(n)$ plus the input size. Formally: $n + \max_i\{p_i\} \leq M_{total} \leq n + W(n)$.*

Proof. In the worst case, each $\mathscr{P}_i$ adds an extra p_i of memory use. The Lemma follows from the fact that $W(n) = \sum p_i$. □

We can now focus our attention to the actual time complexity of the translation of a single step.

Lemma 6. *Work-Time CRCW-decoupled parallel step s_i, with p_i available parallelism can be translated into SPE program $\mathscr{P}^{(i)}$ with $O(p_i + a(M_{PH}^{(i+1)}))$ time complexity.*

Proof. Consider the for loop of the implementation. Its body has address dependence depth $D = O(1)$. Therefore, as in Example 1, the IGU is able to roll out $\lceil p_i/k \rceil$ segments with $O(k + a(M_{PH}^{(i+1)}))$ time complexity each. More precisely, at least $\lceil p_i/k \rceil - 1$ segments have $O(k)$ complexity, since $k \geq a(M_{total}) \geq a(M_{PH}^{(i+1)})$. Summing up, we get $O(p_i + a(M_{PH}^{(i+1)}))$.

As for reduction–like CWs, the simulation adds a functional dependence for each concurrent WT operation. Anyway, the internal forwarding mechanism of SPE can deal with them with no additional slowdown. The same holds for CW implementations based on predicated instructions. □

Next we show how Lemma 6 can be repeatedly applied in order to get a whole SPE implementation of $\mathscr{A}_{WT}$. Correctness follows from the fact that each single application produces the same memory state transition as the correspondent parallel step.

Theorem 7. *Consider WT algorithm $\mathscr{A}_{WT}$, with W work and T time complexity. Then an equivalent SPE program $\mathscr{P}_A$ can be written, with complexity $O(W+T\tilde{a}) = O(W+Ta(M_{total}))$, where $\tilde{a}$ is the average of the worst case memory access latency of each step, $\tilde{a} = \frac{1}{T}\sum_{i=0}^{T} a(M_{PH}^{(i+1)})$.*

Proof. $\mathscr{P}_A$ can be obtained with T applications of Lemma 6. The resulting complexity is therefore $\sum_{i=O}^{T-1}(p_i+a(M_{PH}^{(i+1)}))$, which is $O(W+T\tilde{a})$. Since $a(M_{total}) \geq \tilde{a}$ is always true, $O(W + Ta(M_{total}))$ is a less strict upper bound to the complexity. It is nevertheless usefull as it is much easier to derive in actual applications. □

One should note that this simulation results in a program of $O(T(n))$ instructions. Therefore instruction memory latencies can be ignored.

Corollary 1. Let $\mathscr{A}_{WT}$ be a work–optimal WT algorithm. If $Ta(M_{total})$ is $O(W)$, then there exists a SPE implementation of $\mathscr{A}_{WT}$ with optimal RAM complexity.

Proof. Direct application of Theorem 7. □

Note that a stronger result is also valid, for $T\tilde{a} = O(W)$. It is more difficult to derive in actual applications. In general, any work–optimal parallel algorithm with polylogarithmic time complexity is a good candidate for efficient implementations on the SPE, if $a(x) = o(x)$.

Anyway, another metric emerges from Prop. 5. In fact, the exploitation of the available parallelism, besides helping overlapping latencies, also increases memory use. Actually, once the condition $T\tilde{a} = O(W)$ is met, any further parallelism would just increase memory footprint.

The example of matrix multiplication in Section 2.5 clearly outlines this mechanism.

2.5 Applications

Consider $\mathscr{L}$, the set of programs consisting of *nested for loops*, where the only branches are those determining the execution of loops, and $\mathscr{L}_D$, the subset of $\mathscr{L}$ with address dependence depth at most D. As demonstrated in [4], a program $\mathscr{P} \in \mathscr{L}_D$ can be executed by an SPE with $k = a(M)$ stations in time $T = O(D(N + a(M)))$, where N is the length of the instruction stream of $\mathscr{P}$ on a particular input, and $a(M)$ is the maximum memory latency. This proves the efficient execution of a wide class of algorithms, including matrix multiplication, FFT, etc., whose address dependence depth is $O(1)$.

Other sequential algorithms, which target important, basic problems, have a less regular structure, and are not directly covered by the previous result. For example the problem of merging two sorted lists of n elements has a $O(n)$ address dependence depth. An implementation with constant slowdown with respect to the RAM lower bound was left as open problem in [4]. In general, we have to pay particular attention when implementing RAM algorithms which use data structures such as queues or stacks. In fact, if there are not enough operations to amortize latencies on, programs will exhibit non-constant slowdown with respect to RAM complexity.

For these problems, if a work–optimal WT algorithm exists, with polylogarithmic number of parallel steps $T(n)$, Corollary 1 can be successfully applied to obtain an optimal SPE program, whenever the memory access function of the PHM is $O(x^\alpha)$, $\alpha < 1$.

Connected Components of Dense Undirected Graphs The *Connected Components* problem for a dense, undirected graph $G = (V, E)$, $|V| = n$, $|E| = m = O(n^2)$, consists in finding the connected subsets of vertices in G. It can be sequentially solved in $O(n + m)$ in a RAM, using either breadth-first search or depth-first search [11]. The adaptation of this algorithm to SPE is non-trivial,

and it would require the reorganization of computations to amortize latency. In [11], a parallel solution to the problem is presented, which exhibit $T(n) = \log^2 n$ and $W(n) = n^2$, where input size is $O(n^2)$. Therefore, applying Corollary 1, we can implement a program for SPE with $O(n^2 + \log^2 n \cdot a(n^2)) = O(n^2)$ complexity, which is optimal when $|E| = \Theta(n^2)$.

Merging two sorted lists The classical algorithm for merging 2 sorted lists of n integers each has $D = O(n)$ address dependence depth. In fact, the operands of each comparison depend on the outcome of the previous one. Memory access pattern, in other words, heavily depends on the input, thus making it hard to optimize input layout in memory. Moreover, there is almost no computation to amortize latencies on.

A standard approach for mitigating latency overheads in machines with block-transfer capabilities is *prefetching*; by prefetching, one tries to cheaply move the data which is likely to be used next into the fastest memory locations One way to accomplish this consists in logically partitioning memory into contiguous levels, and moving blocks of items among them instead of single items. By appropriately choosing the size of the levels and of the blocks it is possible to amortize the cost of inter-level exchanges over the number of moved elements. Anyway, such technique is not by itself sufficient to achieve a linear time SPE program for merging.

It is possible, on the other hand, to resort to the work–optimal algorithm in [13], which merges two sorted lists of n items in $T(n) = \log n$ parallel steps, and has $P(n) = W(n)/T(n) = n/\log n$ available parallelism. Direct application of Corollary 1 leads to $M(n) = O(W(n) + n) = O(n)$ memory occupation. The resulting time complexity is $O(W + T\tilde{a}) = O(W + Ta(M)) = O(n + \log n a(n))$, which is therefore linear.

Mergesort It may seem trivial to apply the result on merging as a subroutine for mergesort. It must be noted, though, that the running time analysis for merge relies on the assumption that all the input is stored in the fastest n memory locations. If such assumption is not met, some extra thought is needed to cope with the higher latencies. Consider for example an iterative bottom–up implementation of mergesort for the SPE.

At iteration j, $0 \leq j < \log n$, we have to merge pairs of 2^j-sized lists, with the ith pair stored in memory from position $2^{j+1}i$ on. Each such merge has a $O(2^{j+1} + a(2^{j+1}i))$ complexity, which is superlinear if $2^{j+1} < a(2^{j+1}i)$. As an example, if $a(x) = x^\alpha$, the overall complexity of step $j = 0$ is $\Omega n^{1+\alpha})$.

In this case, a technique similar to the execution of consecutive searches of [14] can be employed. Basically, instead of merging one pair of sublists at a time, whenever the size of the subinstances is small enough, all the merges advance "con-

currently". In other words, the time intervals in which distinct pairs are merged are interspersed in a round-robin fashion. Therefore, since distinct subistances are independent, it is possible to obtain segments of independent instructions, which can be executed efficiently.

Pattern Matching Algorithms The *Pattern Matching* problem consists of finding a given m-sized pattern on a n-sized string, both defined over the same alphabet. Several algorithms optimally solve in $O(n+m)$ operations this problem in RAM model, for example the Knuth-Morris-Pratt algorithm [12].

Similarly to the merge of lists, a straightforward execution of the RAM algorithm would not be efficient in the SPE. In fact, both the failure function creation and the search algorithm present a loop with constant-sized body, where operands depend on an indirect memory access whose address is computed in the previous cycle. The algorithm presents an $O(n)$ address dependence depth, while there are few operations to amortize the latency of memory accesses, leading to an $O(a(M) \cdot (n+m))$ complexity.

In parallel computing, optimal algorithms are known, with $W_{n,m} = O(n + m)$ and $T_{n,m} = O(\log m)$, for example the one described in [11]. We can apply Corollary 1 to this algorithm, obtaining an optimal SPE program, with $O((n + m) + a((n + m) + n) \cdot \log m) = O(n + m)$ complexity.

Matrix Multiplication The well-known matrix multiplication algorithm with three nested for loops exhibits a very marked locality. Besides, it falls within the cathegory of nested-for-loop algorithms which are efficiently executed by the SPE.

Anyway, the application of Corollary 1 helps to shed some light on the memory requirements of concurrency exploitation.

Let us consider, to this purpose, the Work-Time algorithm in which all the n^3 scalar products are executed in parallel, and sums are organized as a tree. Since it is work optimal ($W(n) = n^3$) and requires $T(n) = \log n$ parallel steps, the correspondent SPE implementation $\mathscr{P}$ has $O(n^3 + \log n a(n^3))$ time complexity, which is $O(n^3)$ for $a(x) = o(x)$. The result is achieved even if no locality is exploited, but at an extra memory cost $M = W = n^3$, well above the required $O(n^2)$. In fact, all the products need to be stored in memory at the same time.

It is possible, on the other hand, to bound the amount of exploited parallelism, by recurring to a Work Time algorithm that executes only n^2 products concurrently. In this case, both memory use and running time are optimal, because only n^2 products need to be stored in memory and the available parallelism is enough for amortizing latencies.

Chapter 3

Parallel Limiting Technology Machines

3.1 Machine Organization

In the research for improved sequential execution performance and scalability under constraints of maximum information density and speed, the natural course of action after considering sequential machines is the exploration of designs which explicitly target instruction level parallelism, enabling concurrency exploitation for both memory accesses and instruction execution.

The design which is proposed in this work indentifies two distinct, non overlapping regions of space: one for storage (a main memory) and one for processing (a CPU). Communication between memory and processor takes place across an interface implemented on the shared border. It can be seen as a natural evolution of the SP and SPE processors with pipelined hierarchical memory, in which the main effort is to overcome any sequential behaviour which is not dictated by fundamental constraints.

One should consider the somewhat limited rigor in the former distinction as the processor itself contains a non-negligible amount of memory which is necessary for its activity, mainly in the form of registries and buffers. Still, the main memory is the place where the machine state is represented, at least at certain time instants (including the beginning and end of program execution) and the input and the output are stored, while the processor only holds cached, temporary copies and intermediate values.

This design clearly belongs to the category of machines which exhibit a so-called von Neumann tube (???). As a consequence, all processor-memory communication is subject to the necessarily limited capabilities of their connection interface, which is by all means an artificial bottleneck. However, the somewhat conservative choice

of machines with separate processor and memory allows for approaching more radical designs in a gradual fashion, and identifying more precisely the single changes and challenges which need to be introduced or tackled. The space of possibilities allowed by physically implementable machines is indeed quite vast; it is therefore helpful to both the researcher and the reader to explore and present the findings as a gradual evolution from acquired and shared models and knowledge.

Another, minor, reason to consider machines which exhibit a von Neumann tube, is that actual general purpose machines are not likely to abandon this organization any time soon.

Reasons for this include on one side the vast heritage of software, algorithms, models and know-how which has accumulated over the years, and on the other the difference in production technology between storage and processor elements. Indeed, memory and processor technologies are extremely difficult to integrate on the same chip as the former aims at density and capacitance, while the latter favors switching speed.

It must be noted that there is active research to overcome this state of things (most notably the *iRAM*), but so far did not prove sufficiently cost-effective to make it into mainstream general purpose computing machines.

3.2 A Pipelined, Hierarchical, High-Bandwidth Memory Organization

Design targets The proposed memory design is both hierarchical and pipelined, in order to allow the exploitation of both locality and concurrency of memory accesses during program execution.

As it has been show in [4], even if they are not in general sufficient, both features are necessary in order to be able to hide the effect of latencies with larger machines and problem instances, and to achieve scalability.

In fact, when considering the performance of a non-hierarchical memory system for asymptotically larger and larger machines, it must be noted that the time for each single access would grow with the size of the memory. More precisely, being d the number of dimensions in which the memory circuits are laid out and m the storage size, latency would be at least $\Omega(m^{\frac{1}{d}})$. The only way to amortize latencies would then be by overlapping accesses, but again, the required concurrency would need to match the growing latency.

As a consequence, the more sequential, constrained portions of any program would end up dominating the overall execution complexity, their effect being proportional to latency.

More formally, let *seq* be such portion of program, for which accesses do not

exhibit any concurrency, that is, there exists T_{min} such that $\Omega(T_{min})$ is the least time complexity for *seq*. Such T_{min} could for example arise from a series of dependencies among instructions, each of which require a distinct lookup in memory. It is reasonable to expect that, even considering larger and more powerful machines, the overall complexity of the program would be $T = \Omega(T_{min})$. However, with a non-hierarchical memory system, complexity would be even worse: $T = \Omega(m^{\frac{1}{d}})$, that is growing with the size of memory.

For an asymptotically growing non-pipelined memory system, on the other hand, the difficulty is related to the cumulative effect of larger and larger access times as programs are given larger problem instance to solve. In this case, it would be the portion of the program which exhibit the least locality to ultimately constrain complexity the most. Also the complexity of simpler programs, however, would be severely burdened. Consider, as an example, a program which needs to load into the processor every bit of its input at least once. Then, its complexity would not be smaller than the sum of each single memory access latency. Letting $a(\cdot)$ be the memory access function, that is the function that returns for each memory address, its access time, then the time complexity for *touching* all the bits of an instance of size s_I, is $T = \Omega(\sum_{i=0}^{s_I} i^{\frac{1}{d}})$, which can be rewritten as $T = \Omega(\sum_{i+0}^{s_I^{\frac{1}{d}}} i) = \Omega((s_I^{\frac{1}{d}})^2)$. As can be seen in the formula, the resulting time complexity is much larger then what strictly implied by the larger latency itself.

In order to devise a scalable design for a memory system, it must therefore allow the exploitation of both concurrency and locality.

A hierarchical memory system is also necessary in order to logically shrink the machine. In fact, there are situations when the lack of available locality and concurrency (due to the nature of the program, the implemented algorithm or the problem itself), would make it more convenient to execute at least part of the work with a smaller machine. One such situation may for example occur when a program recurs on smaller instances.

A smaller machine, in fact, is by definition less capable of exploiting e.g. concurrency of accesses, but also has smaller fixed costs (e.g. instruction loading and initialization). With a hierarchical memory system, when the accesses are bound within the first m locations (and consequently the fastest ones), the worst case latency can be bound to $a(m)$, independently from memory size. Therefore, the access cost is the same as the one incurred with a smaller memory system.

In the spirit of [4], let us formally define a *high-bandwidth, pipelined, hierarchical memory*.

Definition 12. A *high-bandwidth, pipelined, hierarchical memory* is a memory system which satisfy the following properties.

- size: M locations with distinct addresses

- memory access protocol: request packets RPs (read or write)
- access function: hierarchy, pipelined period, bandwidth
- B input and output ports uniformily placed on one side
- a RP enters one port and exits *another* port
- each port service 1 RP per clock cycle

order of locations based on latency
a location holds a *word*, which is enough to store an address.
define $m[x]$ as the content of location with address x

The proposed design includes the specifications for a high bandwidth, pipelined non-hierarchical memory (to be used as a building block) and an interconnection strategy (to compose the complete system and connect it to the processor). Each module allows the exploitation of concurrency both by pipelining accesses and by accepting multiple requests per clock cycle. In particular, the interconnection strategy is able to preserve the latencies of the single modules (latency is increased by at most a constant factor) and to aggregate their bandwidth. Finally, by appropriately choosing the size of the modules, it is possible to obtain a hierarchical memory system.

It is also optimal, in the sense that it offers optimal latency and optimal bandwidth at a constant rate (in a sustained fashion).

The resulting design is optimal, in the sense that it offers optimal TODO: complete here

3.2.1 High-Bandwidth Pipelined Memory

The design which is presented here is a non hierarchical, high-bandwidth, pipelined memory of size M, which can be laid out in d dimensions. While two and three dimensional layouts are of practical interest, the design is general enough to be expressed as a function of d.

The goal is to achieve an optimal balance between available bandwidth and worst case latency. In particular, each location has to be reachable from any port with the same latency guarantee. As for the bandwidth, the number of implemented ports is clearly a lower bound, as their operating speed is fixed and limited; however, a high number of ports is not sufficient if they are not in the condition to accept new request packets, e.g. due to congestion issues.

Consider, as an example a memory module laid out in two dimensions. In order to increase the bandwidth, one should favor shapes in which the perimeter is maximized with respect to the area, i.e. a linear array, so to maximize the number

of ports. However, this would have negative consequences on the latency. Indeed, the worst case distance between a port and a memory location would grow linearly with memory capacity. Besides, congestion would cause significant slowdowns in the general case, that is whenever locations are accessed from sub-optimal ports. This, in turn, would prevent the ports from accepting a request packet at each time step, thus curbing the actual available bandwidth.

A balanced design, instead, needs that bandwidth and worst case port to location distance be optimized at the same time. The consequent preferrable layouts occupy a compact d dimensional volume of size M, and communicate through a $d-1$ dimensional surface.

Servicing x request packets would take $T(x, M) = \Omega(x/B + a(M))$ time steps, where B is the available bandwidth and $a(M)$ is the latency. In this scenario, we would have $B(M) = O(M^{d-1/d})$, and $a(M) = \Omega(M^{1/d})$. As a result $T(x, M) = \Omega(x/M^{d-1/d} + M^{1/d})$ which is optimal for any $x \leq M$.

The design itself is based on the principles of modularity and extensibility, and provides an optimal balance between bandwidth and worst-case latency at a sustained constant access rate.

Memory locations are implemented by means of circuits of constant, fixed size. Each circuit, or node, has a fixed storage capacity, ultimately determined by occupied volume of (d dimensional space) and is identified by a univocal address. Word size, that is the amount of bits per location, is large enough to store a memory address. For clarity's sake, and to avoid carrying an additional constant in all the calculations presented, in the rest of this work it will be assumed that there is a one-to-one correspondence between memory nodes and locations. It must be noted that such assumption does not curb in any way the reach of the results.

The fixed volume of nodes, as is the case with storage, also bounds its communication abilities. As a consequence, each node has a fixed amount of interconnections, and through them can send and receive a fixed amount of data per time step.

On the other hand, the choice of circuits with fixed size allows us to bound the time to perform one memory operation to within a constant. It is therefore independent from the size of the memory M, so that the number of nodes in the implementation does not impact on the time for reads or writes.

Memory nodes are connected as a d dimensional array. Such layout only allows for fixed-length interconnections between neighboring nodes. As a consequence, the whole memory can be laid out in d dimensions with optimal volume occupation M, as interconnections require $O(M)$ space.

Moreover, the distance between pairs of nodes, and between the external surface and any node is smaller or equal to the diameter of the array, that is $O(M^{1/d})$.

Finally, the layout offers a $O(M^{d-1/d})$ surface, in terms of number of nodes

which are directly reachable from outside the memory, which is sufficient for guaranteeing the required bandwidth.

The interface with the processor is implemented by means of $O(M^{d-1/d})$ ports evenly spaced on one side of the memory module. The number of ports is proportional to the number of nodes on the surface of the memory.

Like the memory nodes, they offer fixed capabilities, and are implementable with fixed volume occupation. In particular, they have fixed communication and buffer capabilites. On one hand, this is necessary so to achieve a constant access rate, that is, independent of M, the memory size. One the other hand, the consequence is that a port is able to accept a new request packet only if the previously received ones have already been dispatcched towards their destination. Request packets, indeed, cannot accumulate due to the fixed buffer.

To allow for a gradual introduction of concepts with a simpler design, it will be assumed that ports are directly connected to the nodes on the surface of the memory, so that each component has a fixed number of neighbors and all the connections have fixed length.

The worst case distance from the outer interface of a port to any memory node is proportional to $M^{\frac{1}{d}}$, thanks to the fact that ports, nodes and interconnections have fixed size. Thus, worst case access latency is $T(M) = \Omega(M^{1/d})$ which matches the fundamental limits.

Batch Accesses In order to coordinate multiple requests at the same time, memory accesses are organized in *batch*es. Batches are designed so that each one is able to answer M request packets in $T(M) = O(M^{1/d})$ time steps.

One restriction applies, which may seem very constricting. Indeed, for each batch, only up to one memory operation per distinct location is allowed. The key reason for this choice is that the extra management to handle multiple requests on the same location is better suited to be implemented on the processor side of the machine. However, as will be hinted at later in this Section, it is also possible to enrich the memory desing with the ability to manage multiple accesses.

In its simpler version, batch accesses take place one after the other, over and over again. A batch begins with an *ingestion phase*, when request packets are accepted. During the ingestion, each port accepts up to one request packet per time step, and immediately forwards it into the memory, in the opposite direction with respect to the processor. In turn, all the involved memory nodes continue this propagation mechanism in a systolic fashion. As a result, during ingestion, ports are able to operate at full capabilities, that is, at *constant rate*. It must be noted that this is necessary in order for the available bandwidth to match the number of ports. The ingestion phase terminates after $T(M) = M^{1/d}$ time steps, when up to M request packets have been accepted.

Lemma 8. *The proposed memory design is pipelined. In other words, during the ingestion phase of a batch access, ports operate at constant rate.*

Proof. During the ingestion phase, at each time step, the sequence of request packets which have entered one port are shifted in the opposite direction with respect to the processor. As a consequence, in one time step, the port can free its buffers and is able to accept a new request packet. □

After ingestion, ports stop accepting request packets until a new batch is started. For simplicity, it will be assumed that at this point, each of the M memory nodes holds one request packet. The next operation consists in sending each request packet to its specified destination, so that the actual memory operations can be carried out in the appropriate locations. One efficient way to accomplish this task, dubbed *assignment phase*, is to implement an optimal permutation algorithm to sort the request packets with respect to their target memory address. Such permutation algorithms are optimal in the sense that their time requirements is proportional to the length of the array diameter, which is an unavoidable lower bound. Besides, they only require fixed queue size for each node, which is suitable for the design choices of the memory. Once each request packet has reached its destination, each location can execute the specified memory operation. In case of a read, the request packet is populated with the content of the location; in case of a write, the content of the location is overwritten with the value held by the request packet, which are then discarded.

The last step of a batch access is a *reply phase*, in which read values are returned to the processor. As such values do not need to use the same port which accepted the corresponding request, it is sufficient to let them reach the ports in a systolic fashion. In other words, the reply phase is exactly specular to the ingestion, and can be carried out in $T(M) = M^{1/d}$ time steps.

At the end of a reply phase, a new batch can begin, and ports start accepting request packets again.

The total time to service one request packet is the sum of the time spent in the ingestion, assignment and reply phases. Consequently, the latency for a memory operation is $M^{1/d}$, optimal for a memory laid out in a volume M, in d dimensions.

Lemma 9. *The proposed design has optimal latency $T(M) = M^{1/d}$ for a memory of size M. Moreover, latency does not depend from the ports and memory location involved in the memory operation.*

Proof. Let us consider a single read request packet. Once it is accepted by a port, it waits up to $O(M^{1/d})$ time steps for the ingestion phase to terminate. Then, it is routed to its destination in $O(M^{1/d})$ time steps with the assignment phase. Once its target memory node is reached, the actual read can be carried out. Finally, it waits up to $O(M^{1/d})$ time step in the reply phase to reach a port for being output.

The case of a write request packet is identical, with the exception of the reply phase. In fact, once the appropriate memory node is updated, the write request packet can be safely discarded.

In both cases, the total latency is $T(M) = M^{1/d}$ time steps. □

The number of request packets which are serviced for each time step is $O(M^{d-1/d})$. Indeed, each of the $M^{d-1/d}$ ports accepts or replies a request packet per time step, thanks to the pipelined behaviour of the memory. The assignment phase, in turn, routes M request packets in time $T(M) = O(M^{1/d})$, which is equivalent to processing $M^{d-1/d}$ packets per time step.

Lemma 10. *The proposed design provides bandwidth $B(M) = M^{d-1/d}$ for a memory of M locations, which is optimal if the memory is implemented in a compact volume of size $O(M)$.*

Proof. Thanks to the systolic behaviour of the ingestion and reply phases, the ports are able to operate independently at full capacity. Moreover, in the assignment phase, up to M request packets are concurrently routed in time $T(M) = O(M^{1/d})$, which is enough not to slow down port activity. The actual execution of reads and writes, finally, is a completely parallel operation. Therefore, the available bandwidth is $B(M) = M^{d-1/d}$. □

Resulting time complexity for processing a batch of accesses to a memory of M locations is the total time spent in the ingestion, assignment and reply phases, that is $T(x, M) = O(M^{1/d})$. It is important to stress that this complexity result holds for any number of requests smaller or equal to M. In other words, the same, optimal, latency is guaranteed to single accesses as well as to the maximum allowed concurrency.

Lemma 11. *With the proposed design, a memory of M locations can process x request packets in $T(x, M) = O(M^{1/d})$ time steps. The time complexity is constant for any number of request $x \leq M$.*

Proof. The cumulative time for a batch operation, from the first arrival of a request at a port to the instant when the last reply leaves its port, is $O(M^{1/d})$, that is, proportional to the complexity of each of the phases. The same outcome is obtained applying the results for access rate, bandwidth and latency. Putting them into a formula, we have:

$$T(x, M) = O(x/B(M) + a(M)) = O(M^{1/d}).$$

□

Alternative, more refined implementations of batch accesses may let the memory in an idle state when it is not needed. Such a feature could be useful, for example, if the idle state is able to dissipate significantly less power than the active state. In this case, the task of initiating a batch access is let to the processor. It may be accomplished by means of a special activation message sent to all the ports, or even implicitly by starting to send request packets to at least one port. In the latter case, some little extra coordination is needed among the components of the memory, so that all of them are notified the beginning of a batch.

Sustained Access The memory design presented so far achieves optimal average performances. However, the resulting behaviour implies that memory is accessible only during specific time windows, that is, during the ingestion phase.

It is possible, in fact, with a slightly more complex memory organization, to implement a memory which allows sustained access to its locations, without compromising the time and space results obtained so far.

In particular, it is necessary to better decouple the different phases of a batch, so that they can be pipelined.

To this purpose, one solution is to distinguish between input and output ports, and to augment them with suitable buffers. The input buffers, in order to be effective, need to be able to cache the whole content of the ingestion phase. Specularly, output buffers need the same capacity to cache the data to be output. Moreover, both sets of buffers have to be able to exchange their whole content with the memory nodes in a fixed time, so that input buffers become immediately free to ingest new packets, and the memory nodes can receive the new batch of requests immediately after performing the actual reads and writes. Finally, with the addition of these extra structures, latency must not grow by more than a constant multiplicative factor.

One way to fulfill all requirements is to design the buffers as linear arrays of $M^{1/d}$ cells, such that each cell has a fixed storage capacity, and is able to communicate with its neighbors in a fixed, constant time, independent of buffer size. Each linear array is then connected on one end to one distinct port, so that it charges (or dischargers) through it. The linear arrays are laid out in lines perpendicular to the processor memory interface, so that there are no intersections. On the contrary, they are interleaved with the rows of memory nodes, such that each memory node is at constant distance both from a dedicated cell of an input buffer and from a dedicate cell of an output buffer. In this way, it is sufficient to connect each memory node with its dedicated input and output cell for the communication between actual memory and buffers to be fully parallelized, and consequently take just a fixed number of time steps.

Theorem 12. *The design of a high-bandwidth, pipelined memory presented here,*

is able to process x request packets in $T(x, M) = O(M^{1/d})$ time steps in a sustained fashion.

Proof. It must be noted that extra care has been taken so that all the base components which are employed (memory nodes, ports, buffer cells) have fixed size. As part of that, the interconnection topology is a graph with fixed degree (parametric in the dimensionality d), and all connections are near-neighbor, that is, they stretch for a fixed distance which depends only on node size. In order to measure volumes and distances, it is therefore sufficient to count the number of nodes. Consequently, all the performance consideration made for the simpler memory design are valid also in this settings.

On top of this, the buffers are powerful enough to let ports operate at constant rate, so that bandwidth is preserved.

The wide $O(M)$ bandwidth between buffers and the actual memory, on the other hand, prevents excessive latency penalties, and at the same time allows overlapping the assignment phase with ingestion and reply of the next and previous batches. Thus, also latency is preserved. □

Finally, one should note that it would be possible to implement the same mechanism integrating the buffer capabilities into the memory nodes themselves, and interleaving in time the operations for ingestion, assignment and reply. However, the definition of dedicated structures helps better identifying the working mechanism of the memory design, and allows for a much clearer presentation of concepts.

Alleviating the constraint of distinct addresses The proposed memory design requires that each batch contains at most one memory operation per location. Such a restriction may look quite important for a general purpose machine. However, as will be shown in the presentation of the processor, this is not the case. Indeed, the support for multiple requests for location is left as an optional feature because the processor is more suitable for managing intermediate memory operations and avoid unnecessary fetches or commit. Besides, as it has already been shown, allowing one single read or write per location greatly simplifies the routing problem in the assignment phase.

However, in order to allow room for wider applications than the processor presented in this work, it is hinted here how to manage multiple requests on the same location and in the same batch.

Let us describe the problem in more detail. Among accesses targeting the same location, a sequential order is necessary, which is the one expressed by the semantics of the program. It is assumed, then, that each access inherits its priority from the instruction it originates from. An access originating from an instruction which is processed later in the execution will have to be carried out later than

one originating from an earlier processed instruction. This assumption implies the existence of one single stream of instructions, such that each access can inherit its priority from the relative position of its originating instruction in the stream. From another point of view, we assume that in case of parallel programs, the management of parallel accesses is implemented in software, by the program itself, which determines the sequential order of conflicting memory operations.

The solution proposed here is similar in spirit to the one devised in and is an extension of the memory design which has already been presented, in particular of the assignment phase. Actually, this augmentation essentially mimics the behaviour of the processor described in this work, when managing intermediate memory accesses.

The solution consists in extracting from the submitted batch of requests a subset with at most one read and one write per location. Specularly, before outputting the result, the original batch is populated with the correct values. One the hardware side, constant extra storage per memory node is required for hosting the original batch, the size of which is still required not to exceed the size of the memory.

The request packets which are selected, in particular, are the first read and the last write for each memory location, if any. Indeed, the value carried by any previous write would be overwritten. Any further read, in turn, can be given the value carried by the last preceding write, if any, or, otherwise, the value fetched by the first read.

Once the subset of request packets is selected, it can be processed essentially in the same fashion as a batch with no duplicated accesses. The only difference consists in the need for executing a read followed by a write in the same location.

Then, a *post-assignment* phase is necessary to populate the remaining read packets in the original batch. Sorting packets by memory locations and program order can be implemented by means of permutations on the array of memory nodes. With this arrangements, it is possible to efficiently implement the selection; besides, after the selected subset has been processed, each write and each first read can efficiently propagate their value to the appropriate read packets.

Finally, when all read requests have been populated, write request packets can be safely discarded, and the remaining packets sent to the output buffers.

Lemma 13. *A memory of M locations which implements the proposed design is able to process $x \leq M$ memory accesses with duplicates in $O(M^{1/d})$ time steps.*

Proof. It must be noted that the hardware requirements are the same as in the case which has already been debated, to within constant multiplicative factors.

Ingestion and reply phases are not affected in any way; thus they mantain the same complexity.

Selection and post-assignment can be efficiently implemented by appropriately rearranging the request packets, so that the needed communication involve only directly connected nodes over distances smaller than the diameter of the array.

Finally, the assignment phase has the same asymptotic complexity as in the case which has already been presented, since at most an extra memory operation per node must be carried out. □

Further Design Considerations The proposed memory design is non-hierarchical in the sense that the time for a memory access is the same for any location, and is equal to the lower bound access time of the furthest memory nodes.

The achieved performance results, however, match the physical lower bounds. Access rate is constant. Thus, it matches the achievable switching frequency for a circuit of a fixed size. Bandwidth, thanks to the constant access rate, the number and interconnection of the ports, matches the size of the surface of the memory. Latency, finally, thanks to the choice of fixed size base components and the array topology, is proportional to the time to reach the farthest memory location under the constraint of limited signal propagation.

The occupied volume is proportional to the storage capacity, and is determined only by the number M of memory nodes, that is, the design is extensible to arbitrary size only by setting the number of needed locations.

Lemma 14. *The proposed design is scalable to arbitrary size M, in the sense that the described performances apply to any memory size, with no added costs due to increasing size. Moreover, the achieved worst case latency and bandwidth are optimal under constraints of bounded information speed and density.*

3.2.2 A Hierarchy of Memories

The proposed memory design offers a high degree of concurrency, thanks to its ability to accept both pipelined and parallel accesses. However, a necessary feature for a scalable memory system, is a hierarchical organization, that is, an organization which integrates components with different trade-offs between lantency and size. In other words, the number of time steps to access a location must approximate as well as possible the time bound due to minimum distance and maximum signal speed. Formally, the desired memory access function is $a(x) = O(x^{1/d})$.

The key idea is not new (see [??]), and in its simplest version consists in combining memories of distinct size and speed such that they approximate the desired *memory access function*.

However, due to the combined requirements of high bandwidth and minimum device size, the problem of efficiently arranging such non-hierarchical memories, which will be here referred to as modules, takes on some non trivial challenges. In

particular, in order to reach a sufficiently complete solution, it will be necessary to include both sides of the processor-memory connection, and rethink some of the roles in the communication protocol with respect to the mainstream partitioning of tasks.

The first choice to be made, in designing the memory hierarchy, is to decide the size of the modules (the overall number of locations being a direct consequence of it). In order to meet the target memory access function, it is necessary to assign each memory location to a suitable module, and at a suitable distance from the processor. Location x, with target latency $x^{1/d}$, will have to be included in a module with latency up to $c_1 x^{1/d}$, and at distance up to $c_1 x^{1/d}$, for given constants $c1$ and c_2.

Let us for the moment assume that there exists an ideal organization such that the access time to any module is proportional to their intrinsic latency. Let us also assume that this ideal organization is able to fully aggregate their bandwidths. In this conditions, a convenient combination of modules is a set whose linear dimension, the diameters, are in geometric progression (of common ratio $r = 2$, for example). Access latencies follow the same progression, as they are proportional to the diameters. Module sizes are also in geometric progression, but with a larger common ratio, depending on the number of dimensions d of the layout; for a two-dimensional layout, it would be the square with respect to the common ratio of diameters and latencies. Then, locations are numbered such that the addresses that each module gets are integer intervals, and if smaller modules are assigned smaller addresses, so that the resulting memory access function is non-decreasing.

Let us now consider module with index $\hat{k}$ in the sequence, that is the module which contains the memory location referred to by address x. In the current hypothesis, we have that $\hat{k} = \min\left\{k \in \mathbb{N}, \sum_{i=0}^{k} r^{di} \geq x\right\}$, where r is the common ratio of the geometric progression and d the number of dimensions of the layout (for two-dimensional layout and common ratio 2, $r^{di} = 4^i$). Due to the properties of the geometric progression, it is possible to bound $\hat{k}$, and consequently its size and access time, in terms of x. In fact, since $x > \sum_{i=0}^{\hat{k}-1} r^{di} > cr^{d\hat{k}}$, for a constant c, we have that $\hat{k} = O((\log_r x)/d)$. The resulting module size is $O(x)$, with diameter $T(x) = O(x^{1/d})$.

Since the number of time steps to access memory address x is bounded from above by the latency of the hosting module $\hat{k}$, we have that $T(x) = O(diam) = O(x^{1/d})$, which matches the latency due to the bounds on information density and speed.

As for bandwidth, a sequence of c modules in geometric progression provides an aggregated bandwidth of $B = \sum_{i=0}^{c-1} r^{i(d-1)/d}$. Compared with a total size $M = \sum_{i=0}^{c-1} r^{id}$, the aggregated bandwidth is optimal for a memory laid out on a compact region of space, as we have that $B = O(M^{(d-1)/d})$.

Lemma 15. *A sequence of high-bandwidth, pipelined, non-hierarchical memory modules with latencies in geometric progression can be used to layout a high-bandwidth, pipelined, hierarchical memory with optimal memory access function.*

Proof. If addresses are assigned such that the resulting memory access function is non-decreasing, the memory location with address x is hosted by a module with size $O(x)$. Since the inherent latency of the module is bounded by its diameter, location x can be accessed in $T(x) = O(x^{1/d})$ time steps, which matches the latency due to bounded information speed and density.

The aggregated bandwidth B, that is the sum of the available bandwidths of the modules is also optimal for a memory of size M laid out in d dimensions on a compact region of space, as $B = O(M^{(d-1)/d})$. □

Let us now turn our attention to the problem of combining the distinct modules into a pipelined memory system with optimal bandwidth and latency.

The first problem that needs to be tackled is to devise a placement in d-dimensional space such that for each module, the distance with the processor is at most proportional to its diameter. Indeed, for the resulting latency not to asymptotically exceed a module's latency, the wires which connect to the processor must not add larger delays in the communication due to signal propagation itself.

It is immediate to see that such a requirement has a number of non-trivial facets. The first question is the definition itself of such module-processor distance. Indeed, the processor cannot be reduced to a point in space. Since it had not be presented yet, some assumptions are needed to proceed with the design of the memory. The processor has size P, which means that it is laid out in $O(P)$ space in d_P dimensions, where d_P is not required to be the same as d, the number of dimensions the memory is laid out in. It is also assumed that such space is a compact volume, that is, the diameter is $O(P^{1/d_P})$ and the available surface for IO is sufficient to match the bandwidth offered by the memory. Therefore, the latency to access the same memory location is in general dependent on position of the part of the processor which actually originates the request. Even restricting our consideration to the case where the distance is measured from the processor ports is not sufficient to elude the problem. Indeed, due to minimum device size, each port needs a minimum amount of space to be implemented. Besides, in order to guarantee bandwith B, the processor must be equipped with $\Omega(B)$ distinct ports, which in turn have to be (uniformily, if all the available room is employed) scattered in the $d-1$-dimensional surface of the processor, one dimension being required for the connections.

Let us consider now one module of size M_i. Due to these considerations, the time to access its locations is $M_i^{1/d}$ only for at most $O(M_i^{(d-1)/d})$ processor ports. Any further port would necessarily be so far from the module that the time to traverse it would be greater that the module access time.

Remark 1. Memory module i with size M_i is accessible with latency proportional to its access time only within a $O(diam_{M_i} = O(M_i^{1/d})$ radius.

If a processor is laid out in space with diameter $diam_P >> diam_{M_i}$, then only one portion of the processor contained in a $O(diam_{M_i})$ radius will be able to access module i with $O(M_i^{1/d})$ access time.

Note also that if, in the processor, the surface for IO is fully employed for ports layout, then $diam_P$ is also a measure of the maximum distance between two distinct ports.

A specular issue arises with respect to the bandwidth to access. Indeed, if the fastest x memory location are hosted by the smallest memory module which is large enough, then the available bandwidth can not be greater than the one offered by the module itself.

These problems cannot be circumvent as long as both high bandwidth and low latency are required at the same time. Indeed, the implementing the memory-processor interface necessarily faces a space constraints, which are in turn dictated by the fundamental limits of information speed and density.

The approach which is chosen in this work is to reasonably relax the requirements concerning the memory access function and the bandwidth, or, from a different point of view, to more precisely define them.

Let us define the target behaviour of the memory system in terms of accessing a set of distinct memory addresses:

- the available bandwidth depends on the largest of such addresses, which determines the largest module which is involved in the communication;
- latency for accessing such addresses, in turn, is measured with respect to the nearest ports on the processor; again, the largest of such addresses determines the largest module and the largest latency.

In this way, the target time complexity to access the first (and fastest) x memory locations is $T(x) = O(a(x) + x/B(x)) = O(x^{1/d} + x/x^{(d-1)/d})$, which yields $T(x) = O(x^{1/d})$. As it is possible to notice, the target time complexity strikes a perfect balance between the delay caused by the latency and the one caused by the bandwidth.

The proposed organization arranges the set of memory modules with diameter in geometric progression as a skewed pyramid, such that the smaller modules are closer to the processor, and consecutive modules in the progression are adjacent in space. With this arrangement, the distance between a module and the processor is kept at most proportional to its diameter, so that the time for the signals to travel is comparable with the intrinsic latency of the module. It is apparent, however, that for each module but the smallest, part of the surface towards the

processor is obstructed by the smaller modules, and consequently it is not possible to provide the full bandwidth. Anyway, the unobstructed fraction of the surface is still constant with respect to the total, so the resulting bandwith remains optimal.

One of the key simplification in this memory system is that the actual connection between processor and memeory ports are implemented with simple wires. The immediate consequence of this choice is that the task of delivering a request packet to the appropriate memory port, or, more precisely, to the appropriate memory module, becomes a duty of the processor, which has to be carried out before the request leaves the processor port. In fact, the resulting connection results in each module having a separate bus which links it to a dedicated set of ports on the processor.

The choice is motivated by the fact that it is possible to devise a processor which can efficiently accomplish such routing task. For example, in case of a multidimensional array implementation, the required time would be proportional to the diameter, and consequently to the worst case distance between two processor ports.

Any more complex, intermediate routing circuit would not perform better. On the contrary, it could only add further latencies. With the proposed design, all the available bandwidth is already exploited, and the access latency is already pushed to the limits due to signal propagation speed.

Let us formally present the properties of this memory design.

Lemma 16. *The access time $a(x)$ for accessing memory location with address x from the closest processor port is $O(x^{1/d})$, which is equal to the access time $T_i = M_i^{1/d}$ of the hosting memory module i.*

Proof. As implied by Lamma (15), the location identified by address x is hosted by memory module i, with size $M_i = O(x)$. Hence, the module latency is $O(M)i^{1/d}) = O(x^{1/d})$.

The distance between module i and the nearest ports on the processor only depends on the sum of the diameters of the modules which are smaller than i. Due to the properties of geometric series, such sum is proportional to the diameter of module i, that is

$$length(i) = O(\sum_{j=0}^{i-1} M_j^{1/d}) = O(\sum_{j=0}^{i-1} c^{j/d}) = O(c^{i/d}) = O(M^{1/d})$$

which implies that propagation time matches its inherent latency. □

Lemma 17. *The aggregated bandwidth $B(M)$ provided by the proposed memory system, M being its size, is the sum of the bandwidth $B_i(M_i)$ of the single memory modules. Moreover, $B(M)$ is optimal for a memory laid out in volume M.*

Proof. For each module i, the surface facing the processor is proportional to $O(M_i^{(d-q)/d})$. The portion of such surface which is obstructed by smaller modules is proportional to $O(M_{i-1}^{(d-q)/d})$. The ratio is

$$\frac{1}{r^{(d-q)/d}} = constant$$

r being the constant ratio in the geometric progression and d the number of dimensions of the layout. Since such ratio is constant and smaller than 1, the unobstructed surface is still optimal, and the number of memory ports which can be connected to the processor are proportional to the size of the external surface of the module. Hence, the total processor-memory bandwidth is the sum of the bandwidth provided by the single modules. □

More on the choice of module size, and number The proposed module organization can be adopted also with other choices of module sizes. However, it must be noted that the achievable memory access function and the aggregated bandwidth heavily depend on the relative size between consecutive, adjacent modules.

In general, for example, if the diameters of the modules grow more slowly than in a geometric progression, then the latency due to the distance with the processor will (asympotically) exceed the inherent latency of the modules.

Indeed, the geometric progression is a very natural choice for module sizes as it matches the growth of available volume of space within distance k from the processor as k grows, which is another way to show the optimality of the design. In other words, there cannot be more locations with small latency than what is provided here.

The statement is even more constraining if we consider that the most "valuable" volume is needed for both the faster memory and for implementing the connections with the larger, farther modules.

The addition of more small, fast modules to be used as local caches, for example, could be implemented only on a very limited scale just to allow the volume for the connection to the farther modules. On the other hand, this extension would complicate the general access to memory as caches should be kept consistent with the rest of the memory.

3.3 A Processor Organization

We turn now our attention to the processor, which is the portion of the machine in which computing logic is implemented, and consequently, where the actual instruction execution takes place.

The exploitation of the features provided by a high-bandwidth, pipelined hierarchical memory system does require a rather complex processor design.

In fact, it could be said that the processor side of the machine has been overlooked in most of the more used and widespread computational models.

In particular, no processor is able to take full advantage of a pipelined memory if memory requests are not overlapped in time.

3.3.1 Design Goals and Strategy

The goal of this processor design is to provide a scalable architecture able to take full advantage of the capabilities of a high-bandwidth, pipelined, hierarchical memory, and therefore to exploit both concurrency of memory accesses and instruction level parallelism.

In order to do so, it is necessary to fundamentally rethink instruction execution and information flow with respect to the standard processors of models such as the RAM or the HMM. The key assumption to overcome is the fact that the execution of instruction do not overlap in time.

Results in [4] shows that such rethink is already necessary in the context of sequential machines. Intuitively, if an instruction is allowed to submit memory requests only after the preceding one has committed its result, then memory accesses cannot be pipelined and only its hierarchical nature can be exploited through locality.

Such results appear even more substantial when considering models which target instruction level parallelism, and aim at fully exploiting the available high-bandwidth.

The key idea to start from is to distinguish between static programs and dynamic instruction streams. When program $\mathcal{P}$ is executed on a given input, we call the resulting stream of executed instructions a dynamic instruction stream $\mathcal{I}$. In particular $\mathcal{I}$ is referred to as *dynamic* instruction stream because of its dependency on the program input, or, equivalently, the initial memory state.

In classic sequential processors, the instruction stream is generated implicitly during execution, one instruction at a time. In fact, in general, the data which is needed by the current istruction, and the instruction itself to be executed, depend on the result of the previous instruction executions.

The proposed processor takes a different approach and attempts at determining and executing the instruction stream in *segments* of instructions. In order to do so it becomes necessary to resort to features such as branch prediction, in order to speculatively obtain the instructions to include.

Moreover, it should be noted that the definition of instruction stream is rather loose, as in general, for the same program-input pair, more than one instruction stream is acceptable and correct, that is, their executions result in the same final

memory state. In general, as long as the final memory state is unaffected, it is possible to consider different mappings from programs to instruction streams, possibly allowing room for performance optimizations.

Once a segment is generated, the processor tries at the same time to optimize memory accesses and exploit all the available instruction level parallelism. In particular, the relevant portion of the memory state is fetched in one macro access to pipeline the single accesses and to take advantage of the available bandwidth. Moreover, as in general part of the operands becomes available after segment execution has begun, the necessary information flow among instructions of the segment is managed within the processor itself, effectively bypassing memory. In particular, with this approach, functional dependencies can be efficiently dealt with, as instruction results are directly used as operands by the dependent instructions.

In this way, the processor tries to amortize the resulting latency on the time to submit all the memory requests, and on the computational part of the execution complexity of a segment.

The resulting behaviour, generally speaking, is that of a processor which takes a sequence of instructions, eagerly fetches all the needed operands, concurrently computes instructions as soon as their operand are deterministically correct and commits to memory the memory state update resulting from the segment execution.

The optimal segment length is definitely the result of trade-off dictated by the interplay of the different factors which determine performance. Branch prediction accuracy has an impact on the amount of time which is spent in preparing and executing the wrong instructions. Predictions intuitively become more and more difficult as longer segments are required. Locality, that is the slowest memory location accessed in the segment, is another key factor, as longer segments are in general needed to amortize larger latencies. Finally, the available concurrency and parallelism needs to be considered. In fact, in case of (almost) independent instructions, it is desirable to load as many of them as possible, in order to execute them concurrently. In other cases, it may be more profitable to try to exploit locality.

Let us now turn our attention to more quantitative aspects of the processor. The memory design presented in this work provides M locations laid out in d dimensions, accessible with bandwidth $B = O(M^{(d-1)/d})$ and latency for address x described by the non-decreasing function $a(x) = x^{1/d}$. Worst case latency is consequently $a(M) = M^{1/d}$.

In case of a segment of k completely independent instructions, accessing memory locations with address smaller or equal to M, the time for the memory operations is $T(k, M) = O(x/B(M) + a(M)) = O(a(M)) = O(M^{1/d})$. In order not to be a bottleneck, the processor must be able to deliver the requested operands

to the computing nodes and execute within the same time bound. Therefore, its diameter must be $O(M^{1/d})$. A larger diameter would in fact imply larger delays.

The number k of independent instruction can be pushed up to M, which translates into a rate of $O(M^{(d-1)/d})$ instructions per time step, the maximum suported by a memory of size M. The number of functional units in the processor, and consequenltly its size, must be $\Omega(M^{(d-1)/d})$ to keep the same pace. The same figure is obtained for the number of ports needed for communicating with memory.

Let us now consider locality of references, that is the case when the accesses generated by the k instructions are concentrated in the M_k fastest locations. In this settings, the rate is lower, as it is limited by the bandwidth of the memory module which hosts location M_k. The time to fetch or commit $O(k)$ values is indeed $T(k, M_k) = O(k/BM_k + a(M_k)) = O(M_k^{1/d})$.

The bound on the number of functional units is sufficient to deal also with this case, even if k grows up to M_k. There remains, on the other hand, the possibility that k becomes larger still. This may happen, for example, when many instructions request the same data from fast memory. In this case the processor

Let us finally take into consideration dependency management and the internal information flow. Internal processor communication clearly benefits from a low diameter and a high bisection bandwidth.

3.3.2 Processor Organization

The chosen processor organization is a multidimensional array of P constant size functional an communication units, laid out in d dimensions, an analogous configuration to the memory system. The array, or mesh, has indeed useful properties which makes it extremely suitable for designing a scalable and extensible machine. On one hand, multidimensional arrays are bounded degree networks, with only near neighbor connections, which allows straightforward implementations to be consistent with physical bounds, as all the connections are of fixed length. Moreover, extensibility is easily achieved, as the design can be extended with the addition of nodes and the necessary connections.

On the other hand, if node size is a constant, the diameter of the processor grows with $O(P^{1/d})$, while its bisection bandwidth is $O(P^{(d-1)/d})$. Intuitively, this means that the performance of arrays are not compromised by their topology. On the contrary, there exists a vast literature on efficient algorithms for mesh connected processors. In particular, basic operations such as broadcasting, global synchronizations, permutations, sorting, routing, can all be implemented in time $O(P^{1/d})$, which matches the fundamental lower bounds due to maximum information density and limited signal propagation speed and is therefore optimal for any processor of size P laid out in d dimensions.

Let us now focus on the nodes. Each of them is a complete sequential processor, and features all the necessary hardware for instruction execution. In order to be implementable with fixed space occupation, though, such circuits have limited capabilities, which are the ones allowed by fundamental limits on information speed and density in such fixed space.

Storage availability, number of executed instructions per time step, communication capabilities in terms of amount of exchanged data are a function of the chosen node size. It is important to note than, thanks to the bounded degree of the array topology, each node is still able to send and receive a fixed amount of data with all its neighbors in $O(1)$ time steps. Moreover, the maximum number of instructions which can be executed in one time step by the entire processor is proportional to the processor size P. Indeed, among the processors which share this design, the only parameters which identify each single member are processor size P and the number of dimensions of the chosen layout, which are also the only parameters which influence performance.

One important feature is the possibility of using only one portion of the processor as it were a smaller machine. Indeed, as instruction streams are executed in segments of instructions, there are fixed costs which are implied by the management of each segment. For example, sending each instruction to its computing node requires traversing the whole processor, the time complexity for which is in turn at least proportional to $\Omega(P^{1/d})$, the diameter of the array. The same argument also applies to operand acquisition and to writing results back to memory. In favourable cases, such fixed costs, together with the necessary memory access latencies, can be efficiently amortized on the execution time of the segment. Anyway, there are cases for which it is more difficult, if not unfeasible, to amortize such costs. Hence, a better approach is that of reducing them by resorting to smaller segments, and trying to exploit locality of reference by concentrating accesses in the faster modules.

It should be noted that this feature is allowed by a combination of the hierarchical nature of the memory, the processor topology and the structure of the connections between memory and processor.

While on the memory it is possible, if the program exhibit locality of references, to concentrate most of the accesses on the smaller and faster modules, the processor is able to bound its fixed time costs by activating only the square submesh which is directly connected to the faster memory modules.

Therefore, the parameters which need to be considered when executing a segment of instructions are the number of activated nodes in the processor, the largest memory address involved and the number of dimensions of the layout. Combining this with the fact that a subset of the machine can behave equivalently to a smaller machine (i.e. there are no penalty for the larger machine), we have that

the only parameters to consider with respect to executing a segment are the size of processor and memory (respectively P and M), and the number of dimensions of the layout, d.

- considering a 2D memory, a linear array on its side, is enough for that
- however, when we consider also (functional) dependences, it is apparent that the ratio between the size of the segment and the diameter of the processor must be increased; the bandwidth among different subsegments

3.3.3 Program Execution Outline

Let us now turn our attention to how the described machines are able to execute programs. First of all, at a low level, the program can be seen as a bare sequence of machine words which can be stored in a memory. There can be dedicated memories, respectively for data and instructions or a logical partition of a unique memory system. To be executed, anyway, such sequence of words must be accessed and loaded.

Then, an intermediate step is that of generating a segment of the instruction stream to submit to the executing nodes of the processor, based on the current program counter and memory state.

Finally, actual execution takes place which includes accessing memory for operands and final results, assigning instructions to execution nodes and managing the necessary internal information flow.

The following paragraphs outline such stages in details, referring to further parts of this work when necessary.

Instruction flow generation Traditional models as the RAM and HMM implement this step implicitly, as the program is loaded and executed one instruction at a time, and the translation of one program instruction to the corresponding instruction stream segment is trivial. In fact, when proceeding instruction by instruction, all necessary information is always available, and can be fetched along with the instruction operand. The approach chosen in this context is to generate and execute segments of the instruction flow, in order to exploit the concurrency of accesses and amortize latency on the execution time of the segments. Indeed, it is of paramount importance to be able to generate a memory access request before all the previous ones have been answered, so to both increase the use of bandwidth and keep the memory pipeline full.

One big difference with respect to the sequential case is that in the parallel case the execution time to amortize on is in general smaller than the number of instruction themselves.

In general, the instruction flow depends on both the program and the sequence of memory states which are generated as each instruction commits its result. Therefore, it is not in general possible to completely determine the instruction flow prior to the actual execution. It is possible, however, to approximate the list of instructions which are to be executed in the near future, and speculatively generate a limited segment of the instruction flow.

Several techniques and heuristics are already well known and in use, both in actual processors and in the field of compilers, whose aim is that to foresee future evolution of the program counter or future memory accesses. Branch prediction is of paramount importance, as it allows segments not to be constrained by control-flow instructions.

Additionally, a number of optimizations can take place at this stage. As an example, instruction order rearrangement or loop optimizations such as nested loops reordering, loop tiling may be applied to increased the exposed parallelism. Alternatively, or as a complementary approach, it is possible to augment the instruction stream itself with extra directives or annotations, in order to help the processor in the actual execution.

Besides, even more daring approaches may be taken at this stage, at least for specific cases. One could for example decide to execute both ramifications of a branch for a certain length, in order to ensure progress on execution.

All these techniques, however, must be adopted keeping in mind the tradeoff between their beneficial contribution and their fixed costs, or penalty in case of failure (e.g. in case of a branch misprediction).

As a side effect, the ease to speculate correctly on instrucion flow generation becomes a further feature of programs, which adds to characteristics such as locality of accesses and concurrency. This should not be a surprise, as programming techniques such as static loop unrolling essentially aim at producing more easily predictable code.

In this work, it will be assumed that the machine is able to generate segments of instruction flow of a specified length, from a program and an initial memory state. The central topic which is addressed is the execution itself of the generated segment, and in particular the management of internal information flow. If, on one hand, the latter problem is central to the definition itself of the processor, it is also necessary to understand which are the desired features of the instruction flow, that is, to understand which kind of optimization (or augmentation) are the most sensible to implement performance-wise.

Operands acquisition Once the segment has been submitted to the processor, it is necessary to fetch the operands of the instructions for them to be able to produce results.

It should be noted that one step was intentionally overlooked from segment generation to operand acquisition. Such step is in fact the instruction placement on the processor. The reason for this omission is that instruction placement is a key part of any information flow management strategy. Besides, it remains possible to describe operands acquisition without specifying the placement. The only prerequisite is that instructions are placed in the smallest square portion of the processor which is directly connected to the fastest memory modules.

Once instructions are loaded in the processor, it is possible to generate the request packets to submit to memory and fetch the operands. Then, request packets need to be compacted and rearranged, so that the resulting set contains up to one request per distinct memory location, and that the single requests can be submitted to the correct modules through the dedicated ports and connections.

Both operations can be implemented on top of permutations and sorting. In fact, if request packets are sorted with respect to their target memory address, duplicate requests end up clustered in the processor. It this situation it is then trivial to just discard the duplicates.

Then, it is sufficient to permute once again the remaining request packets so that, for each target module, they are placed on a contiguous portion of the processor such that its bisection badwidth is not smaller than the bandwidth of the module.

Once request packets have been prepared, they can be submitted to the memory modules for being processed. Their placement guarantees that processor and memory ports operate at full capacity. As memory modules extract the requested values, the packets are populated and sent back to the processor. Here, an operation specular to the preparation takes place. Indeed, each result must be delivered to each of its requesters. Once again, instructions can generate one request packet for each of its operands, which will include information about the instruction itself. Sorting such packets together with the received values allows to efficiently propagate the correct values to the corresponding requesters. A final sorting is then sufficient to bring back the request packets to the originating instructions.

A slightly more complex case takes place when at least one of the target modules is not directly connected to the active portion of the processor. In this settings it is necessary to activate a larger portion of the processor, so to include also the required ports. This operation has an obvious impact on access time complexity, as the time to access the additional memory ports must be included in the calculations. However, it is also straightforward to see that access complexity is still dominated by the module latency, that is, by the latency of hte slowest accessed location. Indeed, in order to take full advantage of smaller segments it is imperative to bound accesses to the smaller and faster levels, and avoid latency to overgrow the execution complexity of the segment.

In case of indirect addressing, the only change is that the procedure must be executed twice, the first time for retrieving the address of the operand, and the second for the actual operand value.

The results on operand acquisition can now be summed up in the following Lemma.

Lemma 18. *Consider a segment of $n \leq P$ instructions, all accessing locations smaller or equal to M_{max}. Operand acquisition requires $O\left(\max\left\{n^{1/d}; M_{max}^{1/d}\right\}\right)$ time steps.*

Proof. A segment of n instructions need up to $O(n)$ operands to be fetched from memory. Moreover, the minimum diameter of the active portion of processor which is necessary to execute it, is $O(n^{1/d})$. Consequently, all memory modules with diameter up to $O(n^{1/d})$ are directly accessible by the active portion of the processor.

In the case when all accesses are concentrated in these modules, all the requests can be managed by memory in $O(n^{1/d})$. Otherwise, when M_{max} is hosted by oa further module, complexity becomes $O(M_{max}^{1/d})$.

Analogously, on the processor side, the preparation of the request packets and the delivery of results to the requesting instructions is implemented on a fixed number of permutations. The number of time steps for each permutation is proportional to the diameter of the active portion of the processor, which is $O(n^{1/d})$ when only the faster memory modules are accessed. In the case when a slower memory module is accessed, the diameter is proportional to the module diameter itself, which is $O(M_{max})^{1/d}$. □

Remark 2. It is important to point out that for each segment it is suffcient that up to one request per distinct memory location is made. Indeed, there in no performance disadvantage from the side of the processor, in eliminating duplicates and copying the received values. On the other hand, it becomes much easier to guarantee optimal performance of the memory system.

Execution After operand acquisition, at leas part of the instructions are ready to be executed. The hardest part at this stage is the management of dependences, and the consequent information exchange which has to take place within the processor and between processor and memory.

One particularly relevant feature of SP and SPE, in this settings, is *internal forwarding*. Thanks to the ability to propagate computed values to the next instructions, both machines are able to manage functional dependences without any slowdown.

The implementation of SP and SPE heavily relies on the fact that execution is strictly sequential, and the actual communication is a propagation in one single direction of the linear array which implements the processor.

The problem of designing an efficient intra communication strategy becomes much more complex when parallel execution is the target. Indeed, it is the core of this work.

The next chapters will in particular show non trivial lower bounds to this problem, and a series of incrementally improved execution strategies, which end up matching the lower bounds in the worst case.

Chapter improvements

- Figures
  - Mesh
  - hb pipelined mem block
  - ...
- caveat about log size of nodes
- explicitly say that with concurrency memory latency can be amortized
- in general, present the area and problems as in p1-billy
- change/add to the reason for having a pipelined memory: asymptotic analysis needs to take into account that synchronous circuit state changes need to involve bounded regions of space to be scalable, otherwise circuits gets slower with size.
- cite touch problem in the reasons for having a pipelined memory
- definition of hb pipelined hierarchical mem
- cite the iRAM and von Neumann bottleneck, computational RAM (start from wikipedia)
- cite from instruction level parallelism, wikipedia
- explicitly tell the assumption that a system can implement a memory with size proportional to its volume; see "Bekenstein bound" and "limits to computation" in wikipedia

- cite "Routing on meshes in optimum time and with really small queues" by Chlebus, B.S. and Sibeyn, J.F.
- cite HMM

Chapter 4

Lower Bounds on Direct-Flow Programs Execution

This Chapter aims at highlighting the factors which constrain the achievable performance of parallel limiting technology machines.

First, a stripped-down version of the machines which are discussed in this work is presented. For the sake of clarity, only their essential features are considered. In particular, a two-dimensional layout is considered, with size equal to the number of instructions which are to be executed. It should be noted that the hypothesis on the size is allowes by the possibility of selecting a smaller part of the processor for execution, while the restriction on the layout dimensionality does not curb in any way the reach of the results. It is indeed sufficient to adapt the the diameter of the processor based on its layout.

Then, a class of programs is recalled, for which optimal sequential limiting technology machines are already known (see [4]).

It is shown that in the switch to a parallel settings, many factors, which were hidden by the sequential constant execution rate, have distinct contributes to the execution complexity.

In particular, besides processor-memory interaction and intra-processor communication, the topology of the computational directed acyclic graph induced by the dependences of a program is the worst case dominating complexity factor, even in the case when all addresses are known in advance and no further intermediate memory accesses are required during execution itself.

Since these constraints arise from the necessary intra processor communication, regardless of any memory organization there may be, latency hiding is proved to be in general infeasible.

4.1 Machine Model and Direct-flow Programs

4.1.1 Direct-flow programs and related concepts

A dynamic instruction stream $\mathcal{I} = I_1, \ldots, I_N$ is the sequence of instructions which are computed by the processor when program $\mathcal{P}$ is executed on a particular input (or initial memory state).

The pattern of memory accesses issued by the instructions leads to dependences whenever the information produced by an instruction is used by a later one. Such dependences can be conveniently represented by means of a *computation directed acyclic graph*, or CDAG $\mathcal{G}$.

If $\mathcal{P}$ does not contain any control-flow instruction, and only literal or direct addressing is used in specifying instruction operands, then both $\mathcal{P}$ and its (single) $\mathcal{I}$ are direct-flow.

In this case, dependences arise only when the output of an instruction is one of the operands of a later instruction. The case when an instruction produces the address of a later operand cannot take place in a direct-flow program, because of the lack of indirect addressing.

In the context of sequential limiting technology machines, it has been proved in [4] that direct-flow programs can be executed in time

$$T(n, M_{max}) = O(n + a(M_{max}))$$

proportional to the sum of the lenght n of the instruction stream and the single largest memory latency $a(M_{max})$, which is optimal. In fact, it is possible to manage all dependences within the processor, thus avoiding a number of memory reads.

In the context of parallel machines, on the other hand, the importance of direct-flow programs derives from the fact that their instruction streams and the associated CDAGs are static, and do not depend on the input/initial memory state. Therefore, there is more room for optimizations when orchestrating the communications needed to satisfy dependences. Conversely, lower bounds on this restricted, somewhat easier, class hold in general and, as will be shown in this Chapter, are not trivial.

4.1.2 Parallel limiting technology machines

The family of limiting technology machines which is considered in this work are based on an interconnection of basic modules. For each machine, processor and memory are laid out on distinct, non-overlapping areas, which communicate by means of a bus.

A module is an atomic processing or memory unit with fixed capacity. For each clock cycle, the amount of data exchanged with neighbours, its internal memory

size, and the number of computed instructions are constant throughout the whole family of machines. These constraints guarantee that a single module requires only a fixed area to be laid out on.

Since scalability is pursued via the addition of modules, a crucial part of the machine design is the choice of module interconnection topology. For its semplicity, scalability and regularity properties, a square, two-dimensional mesh is the processor layout of choice. The number of modules in the processor is n. The analysis in [5], in particular, shows that a mesh is the best option with regard to scalable limiting technology machines.

Processor-memory bandwidth is proportional to the external perimeter of the mesh, and indicates the number of requests and replies that can be exchanged for each clock cicle. Furthermore, each single memory location is assigned an integer x such that the non-decreasing function $a(x)$ represents its access time.

The next Chapters provide a more detailed discussion on parallel limiting technology machines.

4.2 A general lower bound

Consider the execution of instruction stream $\mathcal{I}$, of length n, on the n-node mesh processor. Let us also assume that $\mathcal{I}$ is direct-flow, that is, all its instructions are of the data-processing type and use either direct or literal addressing modes for their operands.

Interaction with memory does constrain performance in at least two ways. On one side, bandwidth B is limited; on the other, accesss latency $a(\cdot)$ grows with the size of the needed data. Letting n_{acc} be the number of memory accesses made by $\mathcal{I}$, and M_{max} the farthest and slowest accessed location, the time complexity T is at least

$$T(n_{acc}, M_{max}) = \Omega(n_{acc}/B + a(M_{max})). \tag{4.1}$$

As of the processor itself, it is able to execute up to n instructions per cycle, one for each of its modules. In principle, it could execute $\mathcal{I}$ in $O(1)$ time. However, the time to initialize all the modules is at least proportional to the diameter of the mesh:

$$T(n) = \Omega(\sqrt{n}). \tag{4.2}$$

Besides, available parallelism is in general scarse on the instruction stream side, due to dependences. Letting p_{max} be the maximum available parallelism of $\mathcal{I}$, then $T(n, p_{max}) = \Omega(n/p_{max})$. The same line of reasoning leads to a somewhat stronger result. Consider indeed $\mathcal{G}$, the CDAG associated with $\mathcal{I}$. The length of its critical path, L, that is the longest chain of consecutive functional dependences, is the

minimum number of time steps which are necessary to complete $\mathcal{I}$. Therefore, one stricter bound is given by $\bar{p} = n/L$, the average parallelism of $\mathcal{I}$:

$$T(n, L) = \Omega(L) = \Omega(n/\bar{p}). \tag{4.3}$$

The combined effect of processor-memory interaction, intra-processor communication and the available parallelism of the instruction flow leads to the following lower bound:

$$T(n, n_{acc}, M_{max}, L) = \Omega(n_{acc}/B + a(M_{max}) + \sqrt{n} + L). \tag{4.4}$$

Taking into account the fact that B can grow up to $O(\sqrt{n})$ the bound becomes:

$$T(n, n_{acc}, M_{max}, L) = \Omega(n_{acc}/\sqrt{n} + a(M_{max}) + \sqrt{n} + L). \tag{4.5}$$

A further factor must be taken into account, though. Indeed, the interaction of the topologies of both the $\mathcal{G}$ and the mesh processor, and the limited capabilities of each module can easily result in a greater bound than both $\sqrt{n}$ and L.

In the following Section two classes of CDAGs will be defined and analyzed, for which this is the dominant term of the execution complexity.

4.2.1 Comparison with the sequential case

In a sequential processor, instruction execution rate is constant. As a consequence, the length of $\mathcal{I}$ is always able to hide the effect of the limited bandwidth (together with the number of accesses), the lenght of the critical path, and the latency due to the processor diameter.

Besides, in a parallel machine, memory latency is more likely to dominate the overall complexity than a sequential machine, at least if memory access function and processor size are the same.

4.3 Topology-based Lower Bounds

For large classes of CDAGs, execution time on the mesh processor is significantly bigger than the lower bound of Equation 4.5, due to their topology. Two of such classes are presented in this Section, which are based on the binary tree and butterfly graphs. The provided analysis results in constructive proofs of such tighter lower bounds.

4.3.1 Tree CDAGs

The CDAGs which are presented next are based on the binary tree topology. The lower bound is first shown to hold on particular instances of such CDAGs. Then, by means of a simple yet effective technique, the result is extended to a wider class of CDAGs, with almost arbitrary n/L ratio.

Let us consider CDAG $\mathcal{G}_T$, with $2k-1$ instructions, and edges arranged like a binary tree network. Such CDAG will be referred to as *tree CDAG*.

Definition 13. A *tree CDAG* is a CDAG whose topology is that of a binary tree. Dependences induce a partition on $\mathcal{G}_T$ into $\log k + 1$ distinct levels, where k is the number of instructions in level 0. In general, level ℓ contains $k/2^\ell$ instructions.

Lemma 19. *Computing a tree CDAG $\mathcal{G}_T$ of $2k-1$ instructions on a square mesh requires $T(k) = \Omega(\sqrt{k})$ time.*

Proof. Consider instruction s in level ℓ. Its direct and indirect dependences require an area of $\sum_{i=1}^{\ell} 2^i = 2^{1+\ell} - 2$ nodes on the mesh, with diameter at least $\Omega(2^{\ell/2})$. Since data has to flow from any of these dependences to s before s can execute, the time to compute $\mathcal{G}_T$ up to s is at least proportional to the diameter of the area occupied by its dependences, that is $\Omega(2^{\ell/2})$.

Finally, when s is the root of $\mathcal{G}_T$, $\ell = \log k + 1$ and the bound becomes $\Omega(\sqrt{k})$. □

One needs to note that the proof holds also if we allow recomputation on the mesh, because of the constant load constraint.

The above result deals with CDAGs which exhibit a fixed n/L ratio: $n/\log n$. It is possible, though, to concatenate several tree CDAGs, in such a way that the lower bound at Lemma 19 can be extended to the resulting CDAG in an almost straighforward way. Let us define the *repeated tree CDAG*.

Definition 14. A *repeated tree CDAG* $\mathcal{G}_{RT}$ of $n = h(2k-1)$ instructions and with $L = h(\log k + 1)$ is a CDAG whose topology is that of h trees, such that any instruction in the initial level of tree j depends on the root instruction of tree $j-1$.

One key feature of tree CDAGs and repeated tree CDAGs is that their execution is heavily constrainted, and basically the only way to proceed is level by level. In other word, there is a limited number of ways to topologically sort them.

Lemma 20. *Consider the task of computing repeated tree CDAG $\mathcal{G}_{RT}$, with n instructions and functional dependence depth L, on a square mesh. Its time complexity is bounded by*

$$T(n, L) = \Omega\left(\sqrt{\frac{nL}{\log n/L}}\right). \tag{4.6}$$

Proof. Since for each tree in $\mathcal{G}_{RT}$ but the first, no instruction is ready before the previous tree has completely executed, it is possible to extend the bound of Lemma (19) by multiplying it by the number of trees h.

Then, since $n = h(2k-1)$ and $L = h(\log k + 1)$, we have that

$$T(n, L) = \Omega(h\sqrt{k}) = \Omega\left(\sqrt{\frac{nL}{\log n/L}}\right)$$

which concludes the proof[1]. □

Chapter 5 presents an execution strategy for the mesh processor, which is able to execute each level of a CDAG in time proportional to the diameter of the area which hosts it. The resulting complexity is consequently optimal with regard to repeated tree CDAGs. However, it will be shown next that optimality does not hold in general.

4.3.2 Butterfly-like CDAGs

The same technique used for tree CDAGs can be employed with another class of programs.

Let us consider CDAG $\mathcal{G}_B$, with $k(\log k + 1)$ instructions, and edges arranged like a butterfly network. Such CDAG will be referred to as *butterfly CDAG*.

Definition 15. A *butterfly CDAG* is a CDAG whose topology is that of a butterfly network.

$\mathcal{G}_B$ is composed of $1 + \log k$ levels of k instructions each. Computation of one level takes $O(1)$, provided one has enough processors. The most time-consuming activity is communication from one level to the next, that is, to satisfy dependences.

The following lemma shows how the flow of information in $\mathcal{G}_B$ and the topology of the mesh can constrain the time complexity of $\mathcal{G}_B$.

Lemma 21. *Computing a butterfly CDAG $\mathcal{G}_B$ of $k(\log k + 1)$ instructions on a square mesh requires $T(k) = \Omega(\sqrt{k})$ time steps.*

Proof. Consider node x in level $\log k + 1$. In order to its input to be ready, 2 instructions need to be calculated in level $\log k$, 2^2 in level $\log k - 1$ and so on. Therefore $\sum_{j=1}^{\log k} 2^j = \Theta(k)$ instructions have to compute and send their output to x.

Those $\Theta(k)$ instructions need $\Omega(k)$ area (number of nodes) with a $\Omega(\sqrt{k})$ diameter. Hence, since information has to propagate from any of those nodes to x, the time to compute $\mathcal{G}_B$ is $\Omega(\sqrt{k})$. □

[1]The formula has a discontinuity in $L = n$. The donominator should be $\log n/L + 1$.

It is possible to extend the result of Lemma 22 to a wider class of CDAGs. Again, the class is obtained by concatenating butterfly CDAGs.

Definition 16. A *repeated butterfly CDAG* $\mathcal{G}_{RB}$ of $k(h \log k + 1)$ instructions is a CDAG whose topology is that of h butterflies, such that the initial level of butterfly j is the last of butterfly $j - 1$.

Just like tree–CDAGs, butterfly–CDAGs are heavily constrained, with little freedom in instruction execution order. Lemma 22 formalizes the property for butterfly CDAGs.

Lemma 22. *Consider repeated butterfly CDAG $\mathcal{G}_{RB}$. If instruction s in level ℓ of $\mathcal{G}_{RB}$ is ready, and $\ell > \log k$, then at least $k(\ell - \log k)$ instructions have already executed.*

Consider level $\ell \leq \log k$ of $\mathcal{G}_{RB}$. Then, if αk instructions of level ℓ, $\alpha \in O(1)$, are ready, $\Omega(\alpha k \ell) = \Omega(k\ell)$ instructions have already executed.

Proof. All direct and indirect dependences of instruction s of level $\ell > \log k$ are

$$\sum_{i=0}^{\ell - \log k} k + \sum_{j=0}^{\log k - 1} 2^j$$

that is $\Omega(k(\ell - \log k))$.

Consider now αk instructions in level $\ell \leq \log k$. Their direct dependecies are 0 if $\ell = 0$, $\Omega(\alpha k)$ otherwise. Hence the sum of direct and undirect dependences is $\Omega(\alpha k \ell) = \Omega(k\ell)$. □

Lemma 22 states that execution must proceed essentially level by level. For example, let instruction s be in level $\ell = 3 \log k$. Then all instructions in the first two butterflies must execute before s is ready.

Lemma 23. *Consider the task of computing repeated butterfly CDAG $\mathcal{G}_{RB}$ of n instructions on a square mesh. Its time complexity is bounded by*

$$T(n, L) = \Omega\left(\frac{\sqrt{nL}}{\log n/L}\right). \tag{4.7}$$

Proof. Lemma 22 implies that the complexity of $\mathcal{G}_{RB}$ is at least the sum of the complexities of its butterflies. Since $n = kL$ and $L = h \log k + 1$ we have that

$$T(n, L) = \Omega(h\sqrt{k}) = \Omega\left(\frac{\sqrt{nL}}{\log n/L}\right)$$

which concludes the proof. □

Matching the lower bound The general execution strategy of Chapter 5 is not optimal for repeated butterfly CDAGs. In particular, it does not take advantage from the locality offered by the recursive nature of the butterfly topology. The resulting complexity is $T = O(\sqrt{nL})$.

However, the lower bound of Lemma 23 cannot be matched even by an ad-hoc execution strategy. Indeed, as execution proceeds, the area for each smaller butterfly would need to have a low diameter (the square root of its size).

The bandwidth which is required to satisfy dependences in this settings would imply the occupation of a large area of the mesh, which in turn would constrain performance, due to the suboptimal overall diameter to be traversed.

More precisely, the bandwidth needed at level i for a k-wide butterfly CDAG is $B(i,k) = \Omega(2^i\sqrt{k/2^i}) = \Omega(\sqrt{2}^i\sqrt{k})$. The area A on the mesh, which offers such bandwidth while minimizing its diameter d_A, is $A(i,k) = \Omega(\sqrt{2}^i k)$. As a result, the time complexity of executing a butterfly CDAG up to level i is $T(i,k) = \Omega(d_A) = \Omega(\sqrt{A}) = \Omega(\sqrt{2}^i\sqrt{k})$. Considering the last level ($i = \log k$), we have $T(k) = \Omega(k)$.

An ad-hoc execution strategy for the butterfly CDAG Even if the lower bound of Lemma 23 is not achievable, it is possible to improve the $O(\sqrt{nL})$ complexity of the general execution strategy. The key idea relies on a suitable arrangement of the butterfly CDAG on the mesh and on separately iterating on smaller subgraphs in a divide and conquer fashion.

Let us consider, for the sake of simplicity, one infinite mesh. The whole k-wide butterfly is laid out on a $\sqrt{k\log k} \times \sqrt{k\log k}$ square.

The first level occupies a $\sqrt{k\log k} \times \sqrt{k/\log k}$ rectangle at one extremity of the stripe. The second level occupies the adjacent rectangle of the same size. The rest of the stripe is subsequently divided into four equal quadrants, each of size $\sqrt{k\log k}/2 \times (\sqrt{k\log k} - 2\sqrt{k/\log k})/2$.

Then, the layout is recursively applied on the smaller quadrants, as, from the third level on, 4 distinct connected components, which are in turn butterflies, can be isolated.

Actual execution involves two consecutive levels per step. At the end of each step, the output of the second level is sent to each of the next smaller quadrants.

Lemma 24. *A k-wide butterfly CDAG can be executed on a mesh with time complexity*

$$T(k) = O(\sqrt{k\log k}).$$

Proof. The first execution step completes levels 0 and 1, by means of simple data movements on a $\sqrt{k\log k} \times 2\sqrt{k/\log k}$ area. The time complexity for these operations is proportional to the diameter of the area, which is $\sqrt{k\log k}$.

Sending the resulting data to the following level can also be implemented via simple data movements. In this case, the diameter equals the largest dimension of the next smaller quadrants, thus yielding a complexity of $\sqrt{k \log k}$.

In general, step j, which completes levels $2j$ and $2j+1$, takes $O(\sqrt{k/4^j \log k/4^j})$, that is $O(\sqrt{k \log k}/2^j)$. Summing all the steps gives

$$T(k) = O(\sqrt{k \log k}).$$

□

This ad-hoc execution strategy can be easily extended to execute repeated butterfly CDAGs. The resulting complexity is $T(h, k) = O(h\sqrt{k \log k})$, which can be rewritten as

$$T(n, L) = O\left(\sqrt{\frac{nL}{\log n/L}}\right).$$

As a consequence, there remains a $\sqrt{\log n/L}$ gap between the upper and lower bounds. A faster execution strategy does not seem to be easily achievable, due to the limits of the mesh topologies and the fixed capabilities of the modules. It is also not clear if further refinement of bandwidth and diameter arguments may yield stricter lower bounds.

Chapter 5

Upper Bounds - Execution Strategies

This Chapter presents multiple execution strategies for direct-flow segments of instructions. In particular, the problem which is tackled is to implement an efficient intra-processor communication, so to let functional dependences be resolved without accessing memory.

After defining a quite powerful data movement primitive, a number of execution strategies are then presented and evaluated. It is shown, in particular, that extra information, which is a kind of topological partitioning of a segment of instruction, based on its dependences, enables the definition of strategies which can ultimately match the lower bound on intra-processor communication obtained in Chapter 4.

The storage requirements for such extra information, finally, are compatible with the fixed storage availability of each node of the processor.

In order to ease the presentation, only two-dimensional layouts of the processor are considered. Moreover, the processor is assumed to be composed of n nodes, where n is also the number of instructions of the direct-flow segment to execute.

5.1 Avatar Operation

This Section introduces a communication operation on the mesh, which is used throughout the rest of the work. Consider instruction i in instruction stream $\mathcal{I}$. Its essential features with regard to this work are its position within $\mathcal{I}$ and its operands. Its position, which will be referred to as id_i can be used as an identifier. Its input operands, $R1_i$ and $R2_i$, and its output operand W_i specify their role within i, a memory address and an addressing mode. A useful notion is that of instruction *avatars*.

Definition 17. Instruction i of instruction stream *instream* can be decomposed

into 3 *avatars*, each of which is identified by id_i (the position of i within $\mathcal{I}$) and one operand (e.g. $R2_i$).

Note that for each avatar it is possible to define a univocal position within $\mathcal{I}$: it is the concatenation of id_i and the operand position.

The main purpose of an avatar is to represent an instruction with respect to its interactions with memory. An avatar represents what makes an instruction the endpoint of an edge of the CDAG. Moreover, by grouping avatars of different instructions by their memory address, it is possible to identify functional dependences, as for each memory location, a list of all the access is obtained. Let us define a data movement schema on a mesh which exploits avatars.

Definition 18. An *avatar operation* is a mesh data movement schema consisting in the following:

1. decompose each instruction into its avatars;
2. sort by memory location, then avatar position;
3. *communicate*;
4. sort by id_i;
5. piece avatars together into the original instructions.

The time complexity of an avatar operation, with the exception of the *communication* step, is $T(n) = O(\sqrt{n})$ on a square mesh. In fact, sorting can be implemented in $T(n) = O(\sqrt{n})$ time [16], while instruction decomposition and recomposition are embarassingly parallel operations, which take constant time.

It must be noted, though, that one key aspect of sorting is the layout of the elements on the mesh, as an inappropriate choice can easily hinder communication complexity. In particular, it is important to note that the complexity of sorting does not depend on the desired layout (being it row-major, snake, etc.). On the contrary, the choice of the layout is one of the key parameters which determine the complexity of the communication phase.

As for the communication step, its implementation obviously depends on the application of the avatar operation. Let us consider, for example, the task of building the CDAG of an instruction stream. The input is then instruction stream itself; in the output every instruction is augmented, or tagged, with its dependences, i.e. the *id*s of the instructions which produce its operands. It should be noted that only up to two extra instruction ids are needed for each node in the processor, so the extra storage capacity is consistent with the hypothesis on fixed node size.

The following list defines a communication step, which is able to fulfill such task.

1. Detect and isolate submeshes with respect to their avatar's referred memory location.

2. Let each write avatar propagate its id_i to the following avatars, that are neighbour avatars which refer to the same memory address, and whose $id_k > id_i$.

3. Let each read avatar receive id information, keep the maximum, and in case of update, propagate the new value to the following avatars; write avatars do not propagate any message they receive.

It must be noted that, after communication takes place, each read avatar holds the id of the most recent write avatar which is relevant to its memory address. Therefore, once instructions are reconstructed, they hold references to their dependences.

Let us consider the communication step time complexity T_{CS}. As long as avatars with the same memory address occupy a contiguous set of nodes in the mesh, submesh detection can be implemented via a completely distributed test among directly connected neighbours. In fact, the purpose of this substep is to determine which neighbours are relevant for propagation.

As for the propagation itself, one must first note its iterative structure. At each iteration each avatar receives from (and sends to) its directly attached neighbours a constant number of messages. Moreover, since avatars are sorted with respect to memory location, messages do flow in one direction only. Consequently, the time complexity is no more than the diameter of the submesh, which is $T(n) = O(\sqrt{n})$ in the worst case.

Finally, this data movement schema can be easily extended to a d-dimensional meshes with $T(n) = O(n^{1/d})$ complexity. Indeed, both the sorting and the communication phases have complexity proportional to the diameter of the processor.

5.2 An Avatar-based Execution Strategy

This Section shows how the avatar data-movement operation can be adopted to implement the communication, within the processor, which is necessary to fulfill the functional dependences of a segment of instruction stream.

The basic idea is quite simple: it consists in an initialization phase, and the repeated application of the avatar operation. During initialization, the segment of instructions is loaded into the processor and the initial memory state is fetched, while the avatar operations, which are interleaved with a computing phase, implement the actual communication. In the computing phase, instructions are executed if they are ready, that is, when they have received all the needed operands. Each

application of the avatar operation and the subsequent execution will be referred to as *round*.

The readiness test can be implemented by means of a preeliminary avatar operations to detect dependences. Then, whenever a read-avatar is given a new value, it can test whether the originating instruction is the correct one. Instructions which get their operands directly from memory are ready from the start, while the rest become ready to execute as soon as the instructions they depend on complete and communicate their results.

Finally, when all operations have produced their results, the set of write avatars is created, and all the write requests are discarded, with the exception of the most recent for each location. Such selected values are then committed to memory.

Let us define the *avatar-based execution strategy* as the execution strategy defined by the following pseudocode:

1. load segment
2. detect dependences
3. fetch initial memory state
4. while(not completed)
   (a) ready instructions execute
   (b) global avatar operation
   (c) instructions test for readiness
5. filter write avatars
6. commit filtered values to memory

The correctness of the strategy is a direct consequence of the nature of the avatar operation itself, and consists in the equivalency with respect to a sequential execution of the same instruction stream segment. It is in particular possible to give an inductive proof.

Lemma 25. *The avatar-based execution strategy can correctly execute a direct-flow segment of an instruction stream, producing the same final memory state as a sequential execution of the same segment.*

Proof. Let us consider the instructions, within the provided segment, which receive their operands directly from the initial memory state. These, in other words, are the instructions which are not constrained by dependences, and are therefore ready just after the initialization phase. Hence, they execute in the first round. With

the successive avatar operation, all the instructions which only depend on the ones which have already executed receive their correct operands, and become ready. The operands are correct because in a sequential execution of the same instructions, the same values would be produced.

Let us assume that the instructions which have executed up to round $i-1$ have received the correct operands. Consider now the instructions which are ready at round i. Since they all receive the correct operands from the previous rounds, they compute the same results as a sequential execution.

FInally, since for each instruction, operands are either values from the initial memory state or results from previous instructions, at each round at least some of the instructions complete, unless the segment has already been entirely executed. □

It should be noted that, save for fetching the initial memory state, no intermediate memory access is needed, because each instructions receive its operands either from the initial memory state, or from the results of previous instructions, which are delivered exclusively through communication within the processor itself.

Let us now turn our attention to the time complexity of this strategy. With a segment of n instructions, the complexity of each avatar operation is $T(n) = O(\sqrt{n})$ time steps. Also processor-memory interactions, that is, the initial operand fetch and the final commit, do require $T(n) = O(\sqrt{n})$ time steps each. The only remaining variable is then the number of rounds which are performed to execute n instructions.

Lemma 26. *The avatar based execution strategy can execute a direct-flow segment of n instructions in*

$$T(n, L) = O(L\sqrt{n})$$

time steps, where L is the length of the longest chain of functional dependences of the segment, or equivalently, the length of the critical path of the corresponding CDAG.

Proof. Avatar operations and memory interactions do require $T(n) = O(\sqrt{n})$ each. Let us consider the instructions which belong to the longest chain of functional dependences. Then, at each round, one instruction of such chain is executed. Indeed, the first instruction, the only one which is not affected by dependences, executes in the first round. Moreover, the next instruction receives its operands in the same round. Indeed, if one operand would not be available, then there would exist a longer chain of dependences, which would contradict the hypothesis.

In general, by the same argument, for all instructions in the chain, the last operand to become available is produced by the immediately preceeding instruction in the chain.

As avatar operations let each write avatar communicate with the direct dependences, at each round execution completes a new instruction in the chain.

Finally, since the legth of the longest chain of functional dependences is L, the overall complexity is

$$T(n, L) = O(L\sqrt{n})$$

time steps. □

5.2.1 Limits of the avatar-based execution strategy

The avatar-based execution strategy is able to correctly execute a direct-flow segment of instructions, and manages to avoid any memory access (with its inherent latencies), save a first one for initialization and a final one to commit the computed results.

However, such mechanism is quite inefficient whenever the segment is too constrained by dependences, and there is not enough parallelism to exploit. Indeed, the time complexity is superlinear in the number of instructions n whenever $L > \sqrt{n}$. In other words, it can be slower than the sequential computation itself, as both SP and SPE (see [4]) can execute direct-flow segments of instructions in linear time if memory latency is not considered.

The inefficiency of this strategy derives from several factors. The most important is the fact that avatar operations always involve the complete processor, and are not tailored to the number of instructions which are able to compute at a certain round.

Secondly, the fact that $\mathcal{I}$ is in topological order is not exploited. More precisely, the level of instruction i depends only on instructions which preceeds i in the instruction stream.

Finally, no particular layout in the mesh is exploited. If on one hand, this allows the strategy to be quite flexible, and to execute correctly with any instruction layout, on the other hand it strongly hints at the fact that there is room for improvements.

While it is possible to combine the avatar-based strategy with sequential phases in order to avoid superlinear running times, there are other approaches which use extra information (to be introduced next) on the instruction stream to guarantee much better perfirmances.

5.3 Schedules

In order to efficienlty exploit the instruction level parallelism of a sequential instruction stream $\mathcal{I}$, it is convenient to identify a topological partition of $\mathcal{I}$ into

subsets of independent instructions. Such partition will be referred to as a *schedule*, and its subsets as *levels*. The importance of this concept is due to the fact that a schedule allows to encode information about available parallelism in a "ready-to-use" way. In fact, once their inputs are ready, instructions belonging to the same level can be executed concurrently. Moreover, communication can be structured so that it only involves consecutive levels. intuitively, each level can propagate new compute values to the next, so that in the end all the instructions are correctly notified. In the rest of this document, each level will be identified by a unique natural number, and each of its instruction will be tagged with such id. Such information only adds an extra fixed storage requirement per node, which is still compatible with the hypothesis of fixed node size.

Let us then formally define a schedule.

Definition 19. Given program $\mathcal{P}$, a *schedule* is defined as a function $\lambda(\cdot)$ which assigns to each instruction i a natural number, or *level index*, $\lambda(i) = \ell_i$, such that for any functional dependence $i \to j$, then $\lambda(i) < \lambda(j)$.

The similarity with the concept of topological sort is apparent from the definition, as each dependence $i \to j$ is in fact an edge in the correspondent CDAG. The main difference between the two concepts is that a schedule is intentionally looser, to highlight the degree of freedom in executing a sequential instruction stream without changing its semantics, and generally corresponds to more than one linear ordering of the instruction, which can be obtained permuting the instructions of the same level. Also, the definition implies the existence of a lower bound on the number of level indexes, which is L, the lenght of the longest chain of functional dependences, or equivalently, the critical path in the CDAG.

Without loss of generality, it is also assumed that there are no gaps in level numbering, and that the first level index is 0.

Meaningful parameters which characterize a schedule λ are the number of levels L_λ that are induced on a n instruction program, and their sizes n_ℓ, with $\sum_\ell n_\ell = n$.

5.4 Basic Execution Strategy

In the context of nodes of fixed capabilities and near neighbor connections, the exploitation of concurrency, and therefore program execution complexity, are mainly constrained by the need to fulfill dependences, that is, the need to transport data from producers to consumers. One key factor in communication efficiency is the distance between (the nodes which map) dependent instructions. Intuitively, a greater distance directly causes a larger communication time. Moreover, it increases congestion, due to the greater overlap induced by longer paths. These, in fact, need to be mapped to the same wires and nodes. Clearly, though, in a

bounded degree network of processors, such minimization is hard to be met for all dependences, at least in the general case.

The method that outlined in this Section aims at the efficient execution of a generic sequential stream of instructions in the form of a CDAG, while avoiding being unnecessarily complicated. In particular, it aims at minimizing distances among dependent instructions by grouping communications and bounding each group to a small submesh. This is accomplished in the hypothesis that a schedule λ is known in advance and instructions are tagged accordingly.

In order to focus on the basic ideas of this solution, it is also assumed that, being n both the number of nodes in the mesh and the number of instructions to be executed, and L_λ the number of levels induced by λ, each level has n/L_λ instructions, n/L_λ being a power of 4. In the rest of the Chapter, these constraints are gradually lifted to reach a more general result.

With the proposed method, execution proceeds level by level. All instructions in level ℓ complete before any instruction in level $\ell + 1$ starts[1]. Each time a level completes, its output, in the form of a set of write avatars, needs to be propagated to the following levels/instructions via efficient level-to-level communication. This way, dependences which involve consecutive levels are given the highest priority, while "looser" ones are deferred.

From the point of view of level ℓ, the whole process consists in fetching the relevant part of the memory state, i.e. the content of the memory locations referred to by its instructions, at the beginning of the whole execution, and iteratively updating it with the outputs of its preceeding levels, until all the dependences of level ℓ are satisfied. The output of level ℓ, that is, the output avatars of its instructions, is called *memory state update*, and referred to as δM_ℓ.

The generation of one memory state update and its subsequent propagation is called a *round*. The whole execution is composed of L_λ rounds (one for each memory state update), which overlap in time. It is important to note that while different rounds may overlap in time, each instruction is involved in at most one round at a time.

The main goal is to provide guarantees on worst case communication complexity.

The choice of optimizing level-to-level communication, and not every single point-to-point communication goes in this direction. It is, in fact, a more general and simple approach, and, as it will be shown, allows to almost match communication time complexity lower bounds.

It is possible to see, indeed, that the challenging aspect of efficient level-to-level communication is its contrasting requirements. It suffices to consider that the running time T is subject to the following:

[1]Note that this interpretation of the schedule, may be stricter than necessary.

- $T \geq data/bandwidth$, T is inversely proportional to bandwidth;
- $T \geq distance$, T is at least directly proportional to the distance to be travelled (speed = 1 node per time step).

To reflect these constraints, each level-to-level communication is broken down into two steps, which have conflicting requirements: (i) *bulk data transfer* and (ii) *rearrangement* of the received data to match instructions inputs. In fact, while data transfers benefit from an increased bandwidth, rearrangement steps are more efficient if data is not scattered, and distances are bound. Besides, a further sensible requirement is that the time complexity of a level to level communication is independent from the number of levels of the schedule.

The choice for the layout of the instructions on the mesh reflects the need to mediate among such contrasting requirements.

One way to strike a balance is to place the instructions of each level on a distinct, non overlapping, square submesh, and to arrange the levels according to a snake layout, sorted by level index, such that each pair of consecutive levels share an edge. Within each submesh, instructions can be arranged in any order, as besides possibly sharing inputs, instructions within the same level are independent one from each other. We will refer to this layout as the *thick snake* layout.

One should note that, with such layout, the bandwidth between consecutive levels and the maximum distance between any two nodes of consecutive levels are balanced. In fact, on one hand, the bandwidth is proportional to the length of the border that they share, which is $\sqrt{n/L_\lambda}$; on the other hand, the distance between any two nodes is bound by the diameter of the area occupied by the two consecutive levels, $O(\sqrt{n/L_\lambda})$. Therefore, if two instructions are adjacent on a critical path, and are assigned to consecutive levels, the distance between the nodes they are mapped to is $O(\sqrt{n/L_\lambda})$. Finally, the same conditions hold for every pair of consecutive levels.

The method itself is summarized next. An initialization phase fetches from memory all the content of the memory locations needed by the program, assigns them to the relevant instructions, and sorts the instructions according to the thick–snake layout.

Then, the sequence of rounds takes place. As rounds overlap in time, it must be remarked that there is no such thing as a globally synchronized, mesh-wide operation. At the same instant, distinct rounds cohexist in different parts of the mesh. Analogously, the same round is in-flight in different instants, involving different levels. From the point of view of level ℓ, across the sequence of rounds, it is possible to distinguish three states for generic level ℓ:

waiting when the relevant memory state may still be updated by other levels with lower level index;

ready when all the memory state updates from the levels which ℓ depends on have been read by ℓ;

complete when the instructions have been executed, and the relative memory state update forwarded.

Let us recall that we are restricting our scope to the case when only one level is ready in each round. Within round t, only level $\ell = t$ can deterministically (that is, without speculating on some of the operands) execute and produce memory state update $\delta M_\ell = \delta M_t$. If $\ell = t = L_\lambda - 1$, the memory state update is then committed to memory. In the alternative case, δM_t is directly sent to level $t + 1$, which is still in waiting state. Next, δM_t is propagated to the remaining waiting levels $\ell > t + 1$. Each waiting level, after propagating, updates their current relevant memory state with δM_ℓ, and tests itself for readiness. Level $L_\lambda - 1$, finally, commits δM_t to memory.

Once the ready level has triggered the propagation, it becomes complete, and ceases any activity until the next instruction stream segment is provided.

Level $\ell = t + 1$, after propagating δM_t, becomes ready, and starts round $t + 1$.

The following high-level pseudocode formalizes this execution policy.

1. while not complete
   (a) if (waiting)
      i. receive δM_ℓ
      ii. send δM_ℓ
      iii. rearrange δM_ℓ
      iv. test for readiness and possibly set state
   (b) if (ready)
      i. execute and generate δM_ℓ
      ii. send δM_ℓ
      iii. set level state to complete

The same actions are performed by almost every level. The only exception is that level $L_\lambda - 1$, instead of propagating memory state updates, commits them to memory. Level 0, moreover, is ready just after initialization.

The following paragraphs analyze the single steps of the pseudocode and provide possible implementations.

Send/Receive - Bulk Transfer The send/receive mechanism between consecutive levels can be seen as a bulk transfer involving two adjacent submeshes. It can be implemented in a systolic fashion. Each node, as soon as it is ready to send an avatar, tries to contact its neighbor. When the latter is ready, transmission can take place. A number of details need to be settled though. Transmission direction, for example, can be set during initialization, as it is not modified during the whole process. Once all the instructions are sorted according to the thick-snake layout, each node can probe its neighbours for their intruction's level index, and detect the ones which border on their subsequent level. It is then sufficient to propagate such information to the rest of the mesh.

Besides setting the right direction, there needs to be a mechanism for stopping the propagation, a way to detect that the bulk transfer has reached its destination. This can be implemented via a counter (the number of nodes to be traversed can be easily computed during initialization), or by explicitly labelling each avatar with a recipient.

A similar approach can be adopted for memory commits. Indeed, level $L_\lambda - 1$ is responsible of committing all the δM_ℓs. The bandwidth between the submesh hosting level $L_\lambda - 1$ and memory is $O(\sqrt{n/L_\lambda})$. In the assumption that the overall processor-memory bandwidth is proportional to $\sqrt{n}$, the memory is able to accept the δM_ℓs whithout slowing the processor down.

It should be noted, finally, that each δM_ℓ contains up to one write avatar per memory location, and that its avatars can be sorted to further accomodate memory needs.

Data Rearrangement After initialization, all the input avatars in the instruction stream are populated with provisional, speculative values, which refer to initial memory state M_0. Therefore, throughout execution, such prefetched values may need to be updated.

With a bulk transfer it is possible to efficiently propagate memory state updates to all the waiting levels; data rearrangement, on the other hand, is the subsequent process of actually updating the data held by the input avatars.

One way to implement it is by means of an avatar operation involving the input avatars of current level ℓ and the memory state update δM_i. In particular, after sorting all these avatars with respect to their memory location and the snake layout, the output avatars can forward their content to the input avatars with the same memory location. The snake layout implies that avatars which refer to the same memory location occupy a contiguous submesh, which gives guarantees on both correctness and running time.

Finally, the memory state update can be discarded, and the input avatars can be meld back into instructions.

Readiness test In order to execute deterministically, instructions have to be aware of when the operands they hold are consistent with the correct memory state.

If levels are forced to execute sequentially, it is sufficient that, after executing, each level notifies the next, which consequently becomes ready.

This is an artificial constraint, though. In fact, depending on the schedule, a level may be ready for execution before some of its predecessors have completed.

During initialization it is possible to reconstruct the topology of the CDAG, so that each instruction gets information (an id) about the instructions it depends on. Consequently, each time the operands are updated in a rearrangement, it is possible to test if the new data originates from the right instructions. In fact all avatars include references to their instructions. Suppose, then, that each node sets a binary flag to 1 in case of success, 0 otherwise. It suffices to calculate and broadcast the logical AND of such flags to test the entire level.

Memory State Update Generation When a level is ready, all its operands are correct in a deterministic way. By definition of a schedule, all its instructions can execute concurrently, as there is no dependece among them. Finally, as soon as its output avatar is ready, each node begins propagation to the next level.

Correctness In order to prove the correctness of this execution strategy, it is necessary (and sufficient) to show the equivalence with the sequential case. Formally, for any instruction stream $\mathcal{I}$ and initial memory state M_0, this execution strategy has to reach the same final memory state as the sequential execution of $\mathcal{I}$.

The adoption of schedules is at the base of the correctness. Indeed a schedule, by definition, guarantees that for any dependence, the dependent instruction belongs to a level with higher index than the instruction it depends on (level order reflects constraints in dependence resolution). It is therefore safe to:

- concurrently execute instructions with the same level index;
- sequentially execute levels in index order.

There is also a less explicit consequence of relying on schedules. Let us consider the set of memory states reached during the sequential execution of $\mathcal{I}$, and let M_i be the state after instruction i executes. The operands of instruction i, tagged with level index ℓ, can be safely fetched from any memory state M_j in the range $(\ell - 1)n/L_\lambda \leq j < i$. Consequently, all instructions of level ℓ can fetch their operands from memory state $M_{(\ell-1)n/L_\lambda}$.

It remains to prove, then, that the forwarding mechanism of this execution strategy actually delivers the correct operands to the dependent instructions. Let

us introduce the concept of *memory snapshot* of level ℓ at round t, which represents the relevant (with respect to level ℓ) portion of the memory state, updated with the data produced up to round t.

Definition 20. Memory snaphot $\mathcal{S}_\ell^t$ is the set of input avatars of level ℓ after the rearrangement with memory state update δM_t at round t.

$\mathcal{S}_\ell^0$ and M_0 both refer to the memory content before any execution.

It is possible to prove that, for each pair ℓ, t, snapshot $\mathcal{S}_\ell^t$ is consistent with memory state $M_{tn/L}$.

Invariant 1. *For each level ℓ and round t, the operands held by the input avatars of snapshot $\mathcal{S}_\ell^t$ are consistent with memory state M_{tn/L_λ}.*

Proof. An inductive proof is given. The invariant is correct for $t = 0$, as all the snapshots are initialized with values consistent with memory state M_0. It is important to note that the base case applies to all levels.

Let us assume that $\mathcal{S}_\ell^t$ is consistent with M_{tn/L_λ}. As a result, memory state update δM_{t+1} is generated correctly. In the sequential case, memory state $M_{(t+1)n/L_\lambda}$ is the result of executing instructions with level index $t+1$. Therefore, $M_{(t+1)n/L_\lambda}$ can be obtained overwriting the appropriate locations of M_{tn/L_λ} with ${\delta M_{t+1}}^2$.

The bulk transfer, and the thick snake layout guarantee that each level $\ell > t$ receives all the memory state updates, which are delivered in the right order.

Besides, with the rearrangement, each output avatar of δM_{t+1} can communicate with and update all the input avatars referring to the same location. Hence, snaphot $\mathcal{S}_\ell^{t+1}$ is consistent with memory state $M_{(t+1)n/L_\lambda}$.

□

Because of Invariant 1, each instruction is guaranteed to receive the correct operands and produce the correct outputs[3].

Time complexity The whole strategy consists of a sequence of rounds which overlap in time. Consequently, the total running time is given by the time for the first round to complete, plus the number of rounds divided by the round completion rate.

Let us consider a single, isolated round. Its complexity is given by the propagation and rearrangement activities. Execution itself and memory state update generation can be ignored in the calculation, as they are embarrassingly parallel

[2]Note that the single avatars in δM_{t+1} can be applied to M_{tn/L_λ} in any order, as each memory location is affected at most once.

[3]Note that saying that the last snapshot is consistent with the last memory state is not enough for proving correctness. In fact, it is necessary to show that each level commits the correct output to memory, not just the last one.

operations, which take $O(1)$ each, and occour only once per round. Send and receive operations, instead, require moving $O(n/L_\lambda)$ data with $O(\sqrt{n/L_\lambda}))$ bit/clock bandwidth, and for a $O(\sqrt{n/L_\lambda}))$ node distance. Hence, their time complexity is $O(\sqrt{n/L_\lambda}))$. Each rearrangement relies on an avatar operation and some linear propagation. Since they involve a $O(n/L_\lambda)$ node square mesh, the resulting complexity is again $O(\sqrt{n/L_\lambda}))$.

Although the asymptotic complexity is unaffected, it is worth to point out that there is a degree of overlap within the propagation and rearrangement activities. Indeed, each bulk send occour at the same time interval as the correspondent receive. Hence, their complexity needs to be counted only once per propagation step. Moreover, and more significantly, each rearrangement takes place as soon as the preceding propagation moves to the next submesh. As a result, only the last rearrangement is not hidden by the propagation and needs to be considered.

The resulting complexity is $T_{round} = O((1+\tau)\sqrt{n/L_\lambda})$, where τ is the number of bulk send (respectively receive). For the first round, $\tau = L_\lambda - 1$, and $T_{round} = O(\sqrt{nL_\lambda})$.

Let us now focus on the round completion rate. Due to the regularity of the mesh topology, instruction layout and the fixed size of the propagated data, propagation speed is guaranteed to be constant. It is also worth to note, that uniform single instruction execution time is irrelevant in this settings. Therefore, the inverse of the completion rate is the time between two successive bulk send (or, analogously, receive, rearrangement), which comprises a send, a receive and a rearrangement. In asymptotic notation $rate^{-1}_{round} = O(\sqrt{n/L_\lambda})$.

Putting all the components together, the resulting complexity is

$$T = T_{round} + (L_\lambda - 1)rate^{-1}_{round} = O(\sqrt{nL_\lambda})$$

One last caveat is the external loop condition test. In fact, a careless implementation may lead to whole-mesh synchronizations, which would become the dominant component of the time complexity. It is therefore fundamental to reduce the scope of the test to level size submeshes, and allow a complete synchronization only at the completion of the last level.

5.4.1 Design Notes

One key factor in the execution strategy design is to almost completely avoid whole-mesh coordinations or communications. Indeed, it is easy to infer a lower bound from the number x of nodes which need to coordinate, synchronize. Such lower bound is the diameter of the smallest submesh which contains such nodes. As for this execution method, the only coordination among nodes takes place within the same level (readiness test, rearrangement), or within two consecutive

(and adjacent) levels (send, receive), thus preventing the lower bound to affect performance.

It must be noted, though, that node buffer are essential for the implementation of the execution strategy, and obtain the systolic behaviour. In particular, nodes need to be able to store two copies of a memory state update: one to be forwarded on, and the other for the successive rearrangement.

One design alternative is switching the rearrangement and send operations. As for worst case time complexity there is no advantage in doing this, but it is possible to implement a "filter" on the memory state updates. Indeed, if one instruction in a level is going to overwrite a location included in the memory state update, the relative information can be discarded. Note that in this scenario the invariant used for proving correctness does not hold verbatim (due to the discarded avatars). Also, in this case, propagation tends to occur roughly at the same time in the whole mesh. This is slightly harder to implement in the bends of the thick snake, as data has to "change direction". It can be done with constant extra buffer space per node.

As for the choice of sequential level execution, it is by all means an artificial constrain. In general, a level could be ready before some of its predecessor. More precisely, only in the case of the greedy schedule each instruction of a level needs at least one operand from the preceding level [4]. In successive refinements of the execution strategy, it is necessary to explore the effect of allowing both levels and single instructions to execute as soon as they are ready. Again, the correctness proof will not hold with such changes. In fact, Invariant 1 is itself sufficient, but not necessary to guarantee correct execution.

It is reasonable to argue that dependences on the critical path need more priority then the others. A slower than needed information flow in the critical path, in fact, directly translates into poorer performance. This fact is in part reflected by the execution strategy, and in particular by the instructions layout and the propagation mechanism. In fact, critical path dependences are satisfied first, at least if the schedule has not many more levels than the length of the critical path. However, when a new memory state update is sent to the next levels, avatars involved in critical path dependences are not given higher priority. Alternative solutions which give extra priviledge to critical path dependences while avoiding excessive slow downs for the looser dependences, do not seem to be easily achievable. This is in fact an unavoidable consequence of the physical limits on information density and speed.

Finally, it is possible to regard the internal forwarding mechanism of SP and SPE [4] as a particular case of this execution strategy. It suffices, in fact, to choose

[4] If λ has the property that level ℓ is not ready unless it has already received $\ell - 1$ memory state update, it is also true that, *at the same instant*, only one level is ready.

a schedule with one instruction per level. This way, execution would proceed sequentially.

5.5 A Less Constrained Execution Strategy: Variable-Size Levels

5.5.1 Implementation

Here an extension of the basic execution strategy is presented, which lifts the constraints of constant level size. In fact, since such common size must fit all levels, the choice is clearly constrained by the smaller level. A direct consequence is a higher than needed level number, which automatically translates into a more sequential execution and inferior performance. Besides, it is shown that lifting the constant level size constraints is feasible without harming the results obtained in the previous Section, and can be achieved with just an incremental improvement over the simple execution strategy. In conclusion, we keep the complexity result of the simple execution strategy while allowing a much wider range of schedules. This way it is in general more simple to reduce the number of levels.

Still the advantage of allowing variable level size can be pushed further. In particular I will point out the inherent limitation of the instruction layout. Also, it will be shown that in order to further improve performance, it is necessary to switch to slightly more complicated instruction layout and level to level communication.

On the other hand, it is also possible to evolve the variable level size strategy into an online execution strategy with a limited extension (an algorithm to extract the length of the critical path and a smarter fallback when a submesh does not complete in one iteration).

Most of the ideas the simple execution strategy relies on are mantained here, including (with very little modification) the thick snake layout and operand communication (bulk transfers and rearrangements). The main changes involve the work units[5], which are now submeshes, and the instruction execution phase.

Instruction Layout The thick snake layout is (necessarily) slightly modified. We still partition the mesh into L_λ equally sized square submeshes, and we number the submeshes according to the snake layout. The instruction of the first level are placed in the first submesh. If the level overflows, the remaining instructions are placed in the next submeshes, and so on. The only constraint is that a submesh can be populated with instructions only if all the previous ones are already full.

[5]TODO is there a better way to express this? I mean that the pseudocode refers now to submeshes, while with the simple execution strategy it refers to levels.

Next levels are placed in the same way, starting from the first submesh which has not been filled up. As a results, in general levels are split up across more than one submesh. However, the number of submeshes in which a level is split is always $\leq 1 + \lceil n_\ell/(n/L_\lambda) \rceil$. The following Lemma formalizes this claim.

Lemma 27. *With the thick snake layout, the number of submeshes level ℓ is laid out on is $s_\ell \leq 1 + \left\lceil \frac{n_\ell}{n/L_\lambda} \right\rceil$.*

Proof. First note that $n_\ell < n/L_\lambda$ is a sufficient condition for level ℓ not to be split across submeshes only for $\ell = 0$. Therefore, if $\ell = 0$, the number of submeshes is $s_0 = \left\lceil \frac{n_0}{n/L_\lambda} \right\rceil$, which is consistent with the statement.

Let us now consider level $\ell \geq 1$, and let n_ℓ^i be the number of instructions placed in the ith submesh. n_ℓ^0 is the free size of the first submesh which has not been filled up to level $\ell - 1$. Then, the number of submeshes for the remaining $n_\ell - n_\ell^0$ instructions is $\left\lceil \frac{n_\ell - n_\ell^0}{n/L_\lambda} \right\rceil$. Indeed, since only one submesh at a time can be partly filled, all n_ℓ^i, with the possible exception of the first and the last, are equal to n/L_λ.

Summing the submesh which hosts n_ℓ^0, the statement follows. □

Finally, note that the submeshes need not be internally sorted in any way for the property to hold true.

Moreover, as will be shown in the rest of the Section, neither the execution phase relies on any internal layout of the submeshes.

The Variable Level Size Execution Strategy Execution takes again place in a number of rounds. In this context, the rounds involve the submeshes and not the levels.

In details, each submesh can be in the

waiting state when its predecessor has not yet completed;

execution state when the submesh is executing;

complete state when all instructions have executed and their output propagated.

Note that a ready state would not make sense, as it is not guaranteed that all the instructions are ready. Indeed, in general, the instructions placed in a submesh do not belong to the same level; consequently, it is not possible to rule out the existence of internal dependences.

The following pseudocode depicts the execution strategy from the point of view of each single submesh.

1. while not complete
   (a) if (waiting)
      i. receive δM_ℓ
      ii. send δM_ℓ
      iii. rearrange δM_ℓ
      iv. test for readiness and possibly set execution state
   (b) while in execution state
      i. execute and (possibly partially) generate δM_ℓ
      ii. if (execution not complete)
         A. merge the partial δM_ℓ with the level
      iii. else
         A. send δM_ℓ
         B. send activation message to the next submesh
         C. set level state to complete

Readiness test The readiness test is the step which allow a submesh to decide whether or not to begin the execution phase.

A submesh, with the exception of the first, must be allowed to enter the execution state before all its instructions are ready. Not doing so would lead to deadlock, as dependences within a submesh could not be satisfied.

In order to implement the test, it is necessary that the most recently completed submesh notifies the next about its completion, thus allowing execution for the latter.

Relying on the status of instruction operands is not sufficient for guaranteeing the same behaviour. On one hand, when a submesh enters the execution phase, some of its instructions may still be waiting for operands; and on the other hand, all its instructions may be ready before all the preceding submeshes complete.

From the point of view of the worst case scenario, it is not necessary to activate more than one submesh at a time, so it will not be discussed here. However, since it would be beneficial at least in some practical cases, multiple submesh activation will be discussed later in the Section, alongside other enhancements for the execution strategy.

Execution and memory state update generation With submeshes being the unit of work and the possibility that instructions of different levels cohexist, execution is bound to become a multi-step phase. Hence, as illustrated by the pseudocode, it is implemented by means of a nested loop.

At each iteration, only the instructions which hold deterministic operands execute. Their output avatars form the partial memory state update for that iteration, which contains some of the missing input for the unexecuted instructions. More precisely, these output avatars are added to the memory state update generated so far; in case a location is written more than once, the memory state update will contain more than one avatar referring to the same location, each marked with a different instruction id[6]. An rearrangement operation is then used to bring the new information to the instructions which need them.

When at last all the instructions complete, the resulting memory state update is propagated along the thick snake, and an activation message is sent to the next submesh. Finally, the submesh becomes inactive until the next segment of instructions is provided.

Note that, with the possible exception of the first and the last, at each iteration at least a level is completed. And that in a strong sense: there remain no other instructions of the same level to be executed in other submeshes. Indeed, the layout does not allow a level to partially fill a submesh with the exception of the first and the last. As a consequence, if a submesh contains instructions belonging to different levels, then, after execution, all levels but (possibly) the last will are completed. If, on the other hand, a submesh contains only one level (the first), there may be remaining instructions in the next submeshes.

Note also that the fact that at each iteration all the instructions of one level are completed is the worst case guarantee. In particular, unless the schedule is "tight" (i.e. all instructions are assigned in earliest level possible), instructions may become ready before all the previous levels complete, which could be beneficial in at least some practical cases. One should keep in mind, though, that the worst case remains unaffected.

Other enhancements, which may be helpful in practical cases, will be exposed at the end of this Section. In general, they try to better tailor execution units to level size. As will be explained in the next Section, though, there is a fundamental limit which is implied from the layout, which is the fixed level to level communication bandwidth. Without addressing such limit, it is not possible to improve the worst case performance.

Send, Receive - Bulk Transfer The bulk transfer operation is identical to the correspondent phase in the simple execution strategy. Again, its role is to propagate memory state updates along the thick snake, so that levels later on in the schedule will be able to provide the correct operands to the instructions.

[6]This is necessary for example when a level is spread across multiple submeshes and one of its instruction in the first submesh writes to the same location of an operand of a later submesh.

Data Rearrangement As in the simple execution strategy, data rearrangement refers to the update of the speculative operands with the newer values carried by a received memory state update.

However, this operation is in general necessary also within the execution step itself. Indeed, if the instructions of a submesh do not belong to the same level, then its execution will take place in more than one iteration. At the end of each iteration, a rearrangement operation has the task of supplying the non executed instructions with the newly computed values.

The rearrangement operations rely on avatar operations to group avatars referring to the same memory location into contiguous areas of the submesh. Then, within each group, the new values are propagated from its producer to the possible consumers. In order to guarantee correctness, instructions must check that the new values do not originate from an instruction which is due to be executed later in the instruction stream. In this case, in fact, the read avatar has already been given its deterministically correct operand, and has to ignores any further update.

5.5.2 Analysis

Correctness The correctness of this execution strategy can be shown by proving that for each instruction stream, the resulting memory state is identical to the one produced by a sequential execution, even if the intermediate memory state could be quite different.

Schedules are again fundamental, as they guarantee that it is safe to execute instructions in level order, and to propagate the output of a level to the following ones. Hence, since execution follows level order, no instruction is computed before its operands have been computed.

Let us now focus on the first level of a generic submesh. Its operands (the ones which are not directly fetched from memory) are received through bulk transfer operations, which guarantee that each memory state update is forwarded to all the submeshes following the originating one in the thick snake. After the propagation, a rearrangement step ensures that values in the memory state update are matched with the correct input avatars. Since multiple values may be present referring to the same memory location, the match is enforced by checking the id of the originating instruction. Therefore, for each waiting instruction, only the correct operands are set.

Let us now consider the remaining levels of the submesh. Their operands arrive either from bulk tranfers and rearrangements, as in the former case, or by means of further rearrangements within the submesh itself. Since the same test is made to match output avatars with input avatars, instructions are once again guaranteed to receive the correct operands.

Time Complexity As in the fixed-size level case, the whole execution consists of a sequence of rounds which overlap in time. In order to calculate the overall complexity, it is sufficient to consider the intervals between the activations of consecutive submeshes. Unlike the fixed-size level case, though, such intervals are not equal, as they depend on the number of iterations that take place in each execution phase, which in turn depends on the number of distinct levels placed in a single submesh. It will be shown, however, that the complexity in terms of the number of instructions n and the number of levels L_λ remains asymptotically unaffected.

Let us consider the time complexity of a single iteration. Internal rearrangements require $O(\sqrt{n/L_\lambda})$ each, as they rely on whole submesh permutations and linear propagations across up to the length of the diameter. Execution itself and the consequent generation of the partial memory state update are performed in one single step, each node being able to compute independently from the other.

If every executed instruction retains its output avatar until the execution phase is over, then also the assembly of the memory state update can be implemented with (at most) constant overhead. In fact, each node just needs to wait until it is time to start a bulk transfer towards the next submesh.

So, if $L_\lambda^{(i)}$ is the number of different levels in the submesh i, its execution takes $O(L_\lambda^{(i)}\sqrt{n/L_\lambda})$.

Adding the contribution of the initial rearrangement and that of the bulk transfer does not change that complexity, as the two take $O(\sqrt{n/L_\lambda})$ each.

Summing over all the submeshes results in

$$O(\sqrt{n/L_\lambda}\sum_i L_\lambda^{(i)})$$

where $\sum_i L_\lambda^{(i)}$ is the total number of level partitions. As shown in Lemma 27, the number of partition for level ℓ is $s_\ell \leq 2 + \frac{n_\ell}{n/L_\lambda}$. Summing over all the levels yields $O(L_\lambda + l.o.t.)$.

Consequently, the overall complexity is

$$T(n, \lambda) = O(\sqrt{nL_\lambda}).$$

5.5.3 Performance Considerations and Further Improvements

The main goal of this execution strategy is to allow more flexibility in the choice of a schedule with respect to the fixed level case, while mantaining the same achieved complexity in terms of n and L_λ. The performance improvement in this settings follows from the possibility of lowering L_λ given the much wider choice of schedules.

In particular, forcing a schedule to have constant size levels implies a minimum number of levels to preserve correctness. Indeed, this minimum is achieved with a schedule $\overline{\lambda}$ which maximizes the size of its smallest level. Formally,

$$L_{const} \geq \frac{n}{\max_\lambda \min_\ell \{n_\ell\}}.$$

On the other hand, the minimum number of levels in the variable level size is just the longest chain of dependences in the instruction stream. The only way to further lower it is to allow speculative execution and consider an expected time complexity instead of a worst case.

Although it is more flexible with regard to level size there is some stiffness in the mechanism. Indeed, the thick snake layout has a limited adaptiveness in term of communication: both the bulk transfers and the rearrangements. Distinguish between big levels and small levels with respect to submesh size. For large levels, the output bw towards the subsequent levs is clearly too low. Note that the proposed implementation matches this bound. For small levels, if they are not split across two consecutive submeshes, the communication is implemented essentially by means of rearrangements. As a consequence, these rearrangements can be quite inefficient: the involved area can be quite larger than the level itself.

The logical improvement is then designing a more adaptive communication system such that bulk transfers and rearrangements match the size of the involved levels. The results of the next Section are in this direction.

As for the variable level size strategy, several further improvements are still possible, however, which would benefit a range of practical cases. They have not been introduced earlier because they would not improve complexity in the worst case, and they would have somewhat cluttered the essential ideas which are exposed in this Section.

Readiness At the same time, all the instructions which have the correct operands can be allowed to execute. The overhead in managing the added communication is not to be ignored. This enhancement may help, for example, in case of a poor schedule, when instructions are placed in further-than-needed levels.

Eager partial memory state update propagation As the title suggests, the idea is that of anticipating as much as possible the communication of the newly calculated data, so to enable waiting instructions to begin execution earlier. Unfortunately, it is easy to design CDAGs for which this technique would prove useless.

Active submesh shrinking At each iteration of the execution of a submesh, only the instructions which still need to be executed join the expensive operations,

thus effectively shrinking the area involved in permutations. Again, it is easy to design CDAGs for which this enhancement would prove useless.

Recursive behaviour In order to address the inefficiency of the strategy when dealing with small levels, it is possible to recursively define a thich snake in a submesh, and apply the strategy recursively. The case of larger levels is not addressed, unless more radical steps are taken.

5.6 Better Performance: Exploiting Variable-Size Levels

It has been shown earlier in the Chapter that the combination of bulk transfer operations and rearrangements is quite effective in guaranteeing good worst-case performance.

However, the worst case complexity of the variable level size strategy can not be improved unless the inherent limits in inter-level communication bandwidth and latency are addressed.

In particular, all the communication operations need to be taylored to level size. Consequently, the layout itself needs to be able to adapt to larger or smaller levels.

In this Section it is proved that such an execution strategy is feasible. The key ideas of the solution are a modular, recursive layout for instructions and accordingly redesigned data movement operations.

The aim of this execution strategy is to bound complexity to

$$T = O\left(\sum_{\ell} \sqrt{n_\ell}\right).$$

In order to do so, the target complexity for the communication operations between level ℓ and $\ell + 1$ is in turn $O(\sqrt{\max\{n_\ell; n_{\ell+1}\}})$.

5.6.1 Hilbert layout

This execution strategy is still based on bulk transfer and rearrangement operations. Its novel key feature is the layout, which is a new approach to address the problem of bandwidth and latency in the communications between consecutive levels. In particular, the new layout aims at being more flexible than the thick snake, and to better scale with respect to the size of the levels. Whereas the latter provides a fixed bandwidth and a latency which is proportional to level size, the

target for the new layout is $O(\sqrt{\max\{n_\ell; n_{\ell+1}\}})$ for both latency and bandwidth between levels ℓ and $\ell+1$.

Let us define the Hilbert layout.

Definition 21. The *Hilbert layout* on a n-node square mesh consists in placing the instructions of instruction stream $\mathcal{I}$ in level order along the $(n-1)$th approximation of the Hilbert curve.

In this layout, a *module* is the square submesh corresponding to a lower-order approximation of the Hilbert curve.

It is possible to show that such layout provides a number of interesting and useful properties. However, the exploitation of these properties comes at a price, which is an added complexity in the data movement management.

First of all, it is possible to show that with this layout each level occupies a contiguous area in the mesh. Moreover, each such area is compact, that is its diameter is optimal up to constant factors(remember that since each node hosts one instruction, the diameter of an area hosting n_ℓ instructions is $\Omega(\sqrt{n_\ell})$).

Finally, consecutive levels are adjacent in the layout, so that they are able to communicate directly without building up on the congestion or the latency.

Let us start from a quite general result.

Lemma 28. *In the Hilbert layout, any pair of instructions at distance d in the instruction stream $\mathcal{I}$ are hosted by nodes at distance $O(\sqrt{d})$ on the mesh.*

Proof. One direct consequence of adopting the Hilbert curve as a layout is that the first 4^x instructions of the instruction stream $\mathcal{I}$ are mapped to the first 4^x-node module.

Let i, j be two instruction of $\mathcal{I}$ at distance d, that is, separated by d other instructions in $\mathcal{I}$. Consider the smallest module of size larger than or equal to d, which contains i. Such a module exists because of the recursive nature of the Hilbert curve. Then, j may or may not be located in the same module. In the former case, as the module has diameter $\sqrt{(d)}$, also the distance in terms of nodes of the mesh between the two instructions is $O(\sqrt{d})$. In the latter case, it is possible to prove that j is located in the next module (of the same size) along the Hilbert curve. Indeed, the negation of this proposition would contradict the hypothesis that i and j are at distance d. Hence, the distance in terms of nodes of the mesh would be twice the diameter of the modules considered in the proof, which is $O(\sqrt{d})$. □

In order to elaborate further on the properties of the Hilbert layout, let us define induced modules. Intuitively, the modules induced by level ℓ are the portion of mesh which is at least partially occupied by level ℓ.

Definition 22. An *induced module* with respect to level ℓ is a maximal module whose x nodes host at least $c \times x$ instructions of level ℓ, for a constant $0 < c \leq 1$.

Computing the induced modules of a given level can be implemented quite efficiently. Indeed, it is sufficient to loop over module sizes, from 1×1 modules to possibly the whole mesh. At each iteration, the instructions of the relevant level are counted in each module and compared with the target. If the test is passed, then the module is marked as induced. When no larger induced module is detected, the process terminates.

The choice of the constant c determines the number and size of the induced modules of each level. Also, since, as will be shown in this Section, the induced module is the unit of work in this execution strategy, it determines its efficiency.

In particular, by choosing $c = 1/4$, the induced modules of each level are just 2. The following Lemma proves this.

Lemma 29. *For constant $c = 1/4$, the induced modules of level ℓ are at most two.*

Proof. Let us consider level ℓ, with n_ℓ instructions. Let x be the module of size $4^{\lceil \log_4 n_\ell \rceil}$ which contains the first instruction of level ℓ. Clearly, module x is the smallest module large enough to contain the whole ℓ. Note also that its existence is granted by the result on compactness.

We need to distinguish two possibilities: either module x actually contains the whole level ℓ, or part of it "overflows" on the following submeshes.

In the former case, the test on the number of instructions of level ℓ would yield

$$ratio = \frac{n\ell}{4^{\lceil \log_4 n_\ell \rceil}} \leq \frac{n_\ell}{4n_\ell} = \frac{1}{4}$$

and module x would be marked induced. Also, any larger module would fail the test, so x is indeed the maximal module which would be detected.

In the latter case, let n'_ℓ be the number of instructions of level ℓ in module x, and n''_ℓ the number of overflown instructions.

It is now possible to apply twice the argument used in the previous case. As a result, the first n'_ℓ instructions of level ℓ would induce a maximal module of size $4^{\lceil \log_4 n'_\ell \rceil}$. The last n''_ℓ would in turn induce a maximal module of size $4^{\lceil \log_4 n''_\ell \rceil}$.

Each of the two modules completely contains its chunk of level ℓ, so there would not be other induced modules. □

This behavior may look a little overcomplicated, but actually reflects the hierarchy in the recursive nature of the Hilbert curve.

In fact, while it is always possible to decompose a module in submodules of any smaller size, the opposite is not in general true. More precisely, in the worst

case, the smallest common supermodule of two consecutive modules is the whole mesh itself.

By relaxing the requirement that a level is stored in and can be processed by exactly one module, it is possible to circumvent the issue and bound the size of the modules involved.

We can now prove that the set of modules induced by a level have a diameter which is optimal, that is $O(\sqrt{n_\ell})$.

As a direct consequence, it is possible to bound the latency in terms of worst case distance between any pair of instructions from consecutive levels.

Lemma 30. *In the Hilbert layout, the latency, defined in terms of worst case distance between any pair of instructions from consecutive levels ℓ, $\ell+1$ is at most $O(\sqrt{\max\{n_\ell; n_{\ell+1}\}})$.*

Proof. Consecutive levels occupy contiguous areas on the mesh. It is therefore sufficient to sum the diameters of the induced modules to prove the bound. □

The Hilbert layout can be applied also to the fixed level size case. It would result in each level occupying a submesh of the same size of the original version. The only difference would be the ordering of the submeshes. In other words, it would be a different "wrapping" of the thick snake layout.

From this point of view, the Hilbert layout can be seen as a generalization of the thick-snake.

5.6.2 Data Movements

Memory state updates need to move along the Hilbert layout to reach levels with higher index, so that later instructions can be given the correct operands.

However, the variable size of the induced modules is more complicated to deal with with respect to the previous strategies. Indeed, propagating memory state updates along the Hilbert curve means that the propagation speed depends on the size of the memory state update itself. For example an x-node memory state update moves twice as fast as a $x/4$-node one, and this is true also if we do not consider rearrangements. In fact, a module of size n is covered by a n/x node memory state update in x steps if it just moves along the Hilbert curve. Each step takes time $\Omega(\sqrt{n/x})$, leading to a total of $\Omega(\sqrt{xn})$. A single n-node memory state update, on the contrary, would take just $O(\sqrt{n})$. Therefore, with many memory state updates being propagated at the same time, the smaller ones would be outpaced by the larger and faster ones.

Besides, propagation from a smaller induced submesh to a larger one would take too much: $O(\frac{n_{large}}{\sqrt{n_{small}}})$. In order to reach the target performance, propagation between consecutive levels should take $O(\sqrt{\max\{n_\ell; n_{\ell+1}\}})$, or $O(\sqrt{n_{large}})$.

Also broadcasting has to be ruled out. Since it involves a whole mesh syncronization, it would not be possible to overlap different propagations in time, it would be simply too slow to perform for each memory state update.

On the other hand, expanding the memory state update to the largest induced module which is met could seem a reasonable choice. However, an induced submesh in general includes instructions which are to be executed earlier than its own. The propagation process needs instead to avoid contention between propagation and other steps of the strategy.

Also, it would require to compute the induced submesh of all the levels, which is difficult to perform efficiently, as there is in general a significant overlap in space among them.

Memory state updates, during propagation, need to adapt to the possibly larger induced submeshes in another way.

A viable solution is to double the linear dimension of the memory state update each time a module four times its size is covered. For example, a memory state update of size 4^x would cover 4^x nodes in the first step(s), until a complete module of size 4×4^x is covered. At that point the linear dimension of the memory state update would double, thus allowing it to cover the next module of size 4×4^x in one single step.

It is important to note that all memory state updates expand in the same way, regardless of their size. As a consequence, they do not pile up during propagation. In particular, the precomputation of the induced modules is not required.

Besides, this solution does not involve in rearrangements nodes with earlier instructions, as was the case if trying to fit all the induced modules.

Of course, in the implementation, the propagated data has to include also "activation messages", in order to notify the recipients that they are going to be involved in the propagation.

It remains to prove, however, that this solution is fast enough, and that it provides sufficient bandwodth between levels to meet the target for the Hilbert execution strategy.

Lemma 31. *A memory state update covers the first module of size 4^x which follows the generating induced module in the Hilbert curve, in time $O(\sqrt{4^k})$ if 4^k is greater than or equal to its size.*

Proof. The proof is by induction. In the base case, the first propagation step covers the first 4^k-node module, whose size equals the size of the memory state update. The time taken is $O(\sqrt{4^k})$, as it can be implemented in a systolic way with a bandwidth of $O(\sqrt{4^k})$.

Let us now assume that the Lemma holds for size 4^k. Then, by doubling the linear dimension of the memory state update, it is possible to cover the (up

to) three 4^k-node modules in the same fashion in time $O(4^k)$, thus proving the claim. □

It should be noted that the result is quite strong, as it implies that the successor of the currently executing induced module can be fully covered with the memory state update in optimal time whatever its size may be.

Also, in this way each induced submesh can output a memory state update through an edge as large as its diameter, thus meeting the target for bandwith.

5.6.3 The Strategy

Most of the strategy has already been described with the layout and the data movements.

Here, the already described components are plugged together, and the last details are presented.

The strategy is illustrated by the following pseudocode, which describes the whole process from the point of view of a single node in the mesh.

- while $\mathcal{I}$ is not completed
  - if executing
    - start induced module detection
    - execute and produce the memory state update
    - rearrange the memory state update in the induced submesh
    - propagate the memory state update
    - set state to executed
  - else
    - if called in induced module detection
      - partecipate detection and possibly join the induced module
      - partecipate in rearrangement and propagation
    - if called in propagation
      - partecipate the bulk transfer and rearrangement
      - partecipate propagation
      - possibly set state to executing

Induced module detection, propagations and rearrangements are the operations which have been described earlier in this Section.

It may be useful to point out that the doubling of the linear dimension of the memory state update happen with the propagation and that propagation messages need also include information about the size of the module which has to be involved.

As can be inferred from the pseudocode, the only distinction in node state is whether it is executing or not. Indeed, while propagation manages to avoid involving nodes which have already executed, induced modules can include them. Therefore, there is no such thing as a terminated state until the whole instruction stream has executed.

Let us focus on when a node can set its state to executing. Clearly, as there are in general two induced modules per level, and being the single module the unit of work, belonging to a certain module is not enough.

When a level completes, it notifies the fact to its successor in the Hilbert curve. If the successor belongs to the same level, it simply generates its memory state update. Otherwise, the induced modules which are detected next negotiate their execution order based on their relative order in the Hilbert layout. Hence, only the nodes in the first module set their state to executing, while the ones in the second module will stay non executing.

The nodes hosting the first level are set to executing just after initialization, as their operands are directly fetched from memory.

5.6.4 Correctness

The correctness of the Hilbert strategy consists in the equivalence with a sequential execution of any instruction stream $\mathcal{I}$ and initial memory state M_0, with respect to the final memory state M_f.

The proof of this equivalence is based on schedules. Indeed, the execution order of the instructions follows the level numbering in the provided schedule. Therefore, no instruction is executed before its operands are computed.

Let us now focus on the correctness of communication, that is, on proving that instructions receive the correct operands.

Each instruction receive its operands either from memory or from the output of earlier instructions. In the latter case, it is sufficient to point out that all levels receive all the memory state updates generated by earlier levels, as they ultimately cover the whole mesh; and that rearrangements correctly associate output values to inputs, as it does also in the previous executions strategies.

5.6.5 Running time

In order to calculate the running time of the Hilbert strategy, it is sufficient to consider for each level the complexity of the operations between the moment it is

set to executing and the moment the next level become executing. In fact, the rest of the propagation along the Hilbert curve can happen independently from the executing modules, and therefore fully overlaps in time with the other operations.

Induced module detection, as illustrated at the beginning of the Section, can be implemented to run in $O(\sum_{i=0}^{\log n_\ell} 4^i) = O(\sqrt{n_\ell})$.

The initial rearrangement of the memory state update is based on permutations within the induced module, which has a diameter of $O(\sqrt{n_\ell})$. Its running time is therefore $O(\sqrt{n_\ell})$.

The remaining operations which need to be taken into account consist in covering the following level $\ell + 1$ with the newly generated memory state update of level ℓ and the acquisition of the last memory state update from level $\ell - 1$. It can be shown that the complexity for the covering is $O(\sqrt{\max\{n_\ell; n_{\ell+1}\}})$. Indeed, the formula reflects the fact that if n_ℓ is larger, then covering level $\ell + 1$ is completed after the initial propagation step, which takes $O(\sqrt{n_\ell})$.

On the other hand, if n_ℓ is smaller than $n_{\ell+1}$, the propagation strategy exposed with the data movements terminates in time $\sqrt{n_{\ell+1}}$.

The last memory state update acquisition can be dealt with in the same fashion, and accouts for a $O(\sqrt{\max\{n_{\ell-1}; n_\ell\}})$.

For each level, therefore, the complexity to take into account is

$$T(\ell) = O(\sqrt{n_\ell} + \sqrt{\max\{n_{\ell-1}; n_\ell\}} + \sqrt{\sqrt{\max\{n_\ell; n_{\ell+1}\}}}) = O(\sqrt{n_{\ell-1}} + \sqrt{n_\ell} + \sqrt{n_{\ell+1}})$$

where n_x is zero for negative values of the pedix. Summing up over all the levels yields

$$T = O\left(\sum_\ell \sqrt{n_\ell}\right)$$

which meets the target complexity.

Chapter Improvements

- Avatar Operation Section:
  - graphical representation of instruction decomposition into avatars
  - Initialization, including detailed CDAG reconstruction, operands load, both possibly speculative; segment creation could be performed in the unused part of the mesh;
- Schedules
  - graphical representation of a CDAG (note indegree is ¡=2);

  - a (generic) schedule and the greedy schedule
  - graphical representation of the correspondence between a schedule and topological sort (or see next point)
  - show the degree of freedom in a topological sort (e.g. sequence of instructions with arrows-deps, CDAG with schedule; possibly different colors for the different levels)

- Basic Execution Strategy
  - (picture) Thick snake layout
  - (picture) level-to-level trade-offs (max bw: example which does not scale/interspersed; example with max worst case distance)
  - (picture) execution (highlight the overlap in time, but NOT time AND place, of different rounds)
  - (picture) correctness-invariant (think about it...)

- Variable level size exec strategy
  - short intros (analysis, section itself...)
  - complexity: show more equations (e.g. sum of level partitions)

Chapter 6

Conclusions and Future Work

As technology improvements are pushing machine performance against the physical limits and new, diverse, complex architectures are introduced by hardware vendors, the RAM models becomes less and less adequate to assess program execution performances. Indeed, while the latter takes into account only the number of arithmetic logic operations to perform, time complexity is more and more dominated by phenomena of (relatively) high latencies and narrow bandwidths.

Hierarchical and pipelined organizations on one hand, and programs which exhibit locality of reference and concurrency on the other, have proved to be effective countermeasures in large classes of problems.

In particular, the results in Chapter 2 show a metodology for producing code with high concurrency. The resulting code can prove optimal also in problems which are traditionally tackled via the exploitation of locality. Between the two processor introduced by [4], it has become apparent how the different execution strategies can play a big role in the ease of programming and in the memory occupation of such program.

However, sequential machine do not account for all the possible solutions. It makes sense, instead, to investigate parallel organizations, also in the context of the execution of sequential programs.

Chapter 3 to 5 have illustrated a machine design which can scale up to arbitrary size, and can execute a direct flow segment of instruction stream with optimal worst case performance. If the management of instruction dependences is critical for sequential processors, it is even more so for explicitly parallel machines. Indeed, the lower bounds show that functional dependences can be themselves the source of additional complexity, even when memory latencies are not taken into account.

Further steps in the investigation of parallel limiting technology machines should definitely take into account instruction streams with address dependences, for which the dependence graph is consequently not known in advance. The avatar-based execution strategy, despite being in general inefficient, would be already ca-

pable of addressing this challenge. The more optimized strategies, instead, would have to be adapted so to react to schedule mispreditions, multiple accesses to memory for each segment and the consequent time overheads.

Also the computation of instruction stream segments has to be addressed. In particular, schedule computation may be effectively included in this phase, as programs are in general shorter than instruction streams, and may provide more insights in determining the relations between instructions in the instruction stream.

Besides, the effort of exploiting instruction level parallelism could be integrated with the exploitation of thread level parallelism. In other words, it could be possible to decompose the instruction stream and aggregate the instructions in substreams, so to bound the communications among instructions to smaller regions of space.

Then, if it will be proved that no further gains are to be expected from this machine design, the logical continuation of the investigation on limiting technology machines should address designs which do not exhibit a von Neumann tube. Such designs would feature a uniform integration of processing and storage nodes, thus circumventing the need to communicate through a processor memory bandwidth.

Bibliography

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 305–314, New York, NY, USA, 1987. ACM.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 204–216, Washington, DC, USA, 1987. IEEE Computer Society.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computing Conference*, pages 483–485, 1967.

[4] G. Bilardi, K. Ekanadham, and P. Pattnaik. On approximating the ideal random access machine by physical machines. *J. ACM*, 56(5):27:1–27:57, August 2009.

[5] G. Bilardi and F. P. Preparata. Horizons of parallel computation. *Journal of Parallel and Distributed Computing*, 27:172–182, 1993.

[6] Gianfranco Bilardi and Franco P. Preparata. Processor-time tradeoffs under bounded-speed message propagation: Part i, upper bounds. *Theory Comput. Syst.*, 30(6):523–546, 1997.

[7] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

[8] Robert J. Collins. Multiple instruction multiple data emulation on the connection machine. Technical Report CSD-910004, University of California (Los Angeles, CA US), 1991.

[9] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[10] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 89–94, New York, NY, USA, 1978. ACM.

[11] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[12] D. E. Knuth, J. H. Jr. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[13] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput.*, 32(10):942–946, October 1983.

[14] Fabrizio Luccio and Linda Pagli. A model of sequential computation with pipelines access to memory. *Mathematical Systems Theory*, 26(4):343–356, 1993.

[15] Emanuele Milani and Nicola Zago. Exploiting fine grained parallelism on the spe. *ICTCS 2012*, 2012.

[16] Clark D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Commun. ACM*, 20(4):263–271, 1977.

[17] Jeffrey Scott Vitter. External memory algorithms. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings*, volume 1461 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 1998.