Lorenzo Anardu

# M$^2$DF: a Macro Data Flow interpreter targeting multi-cores

Tesi di Laurea Specialistica

Università degli Studi di Pisa
Giugno 2011

Università degli Studi di Pisa
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche
Indirizzo Piattaforme Abilitanti ad Alte Prestazioni

# M$^2$DF: a Macro Data Flow interpreter targeting multi-cores

RELATORE
Prof. Marco DANELUTTO

Candidato
Lorenzo ANARDU

Sessione di Laurea 24 Giugno 2011
Anno Accademico 2010-11

# Abstract

The recent "Moore law crisis" led CPU manufacturers to shift their production to multi-core chips.

Efficient exploitation of such a technology by programmers is a non-trivial issue and requires deep background knowledge.

In order to help programmers some high level libraries and tools based on multi-threading have been developed.

In this thesis we propose an alternative way to efficiently exploit off-the-shelf multi-core chips based on the Macro Data Flow technique.

We describe the implementation of a multi-threaded Macro Data Flow run-time support for multi-core architectures.

The interpreter is assumed to be the target back-end for the compilation of high level, structured parallel programs.

Experimental results are shown on state-of-the-art *Intel*® multi-cores.

# Ringraziamenti

Sarò breve.

Vorrei ringraziare il professor Marco Danelutto per il supporto e l'aiuto datomi durante tutto lo svolgimento della tesi.

Ringrazio anche il professor Luca Gemignani per avermi dedicato il suo tempo ed Alessio Pascucci, sempre presente e disponibile.

Grazie a Giulia per avermi sempre incoraggiato, e per le correzioni stilistiche. Grazie a Giorgio per la supervisione linguistica.

Grazie alla mia famiglia: ai miei genitori per avermi sempre incoraggiato, sopportato e mantenuto, a mio fratello (ormai non più -ino) che mi ha dato per due volte l'opportunità di laurearmi il giorno del suo compleanno (anche se la seconda non la sto cogliendo) ed ai miei zii (con annessi cuginetti) per avermi fatto sentire a casa pur essendone lontano.

Un grazie anche a tutti gli amici che mi sono stati vicini per i bei momenti che mi hanno fatto passare.

Infine vorrei fare un grosso 'in bocca al lupo' a mio fratello per l'imminente maturità, e per tutto cià che verrà dopo.

Sono stato meno breve di quanto pensassi all'inizio, ma va bene così.

# Contents

# Chapter 1

# Itroduction

With improving of manufacturing technology the size of individual gates has reduced, permitting to have more powerful single processor systems. For decades the Moore's law described this trend.

In the early '00 the physical limitations of semiconductor-based microelectronics lead to significant heat dissipation and power consumption. In order to deliver performance improvements, manufacturers have turned to multi-core architectures, also for commodity processors. Actually the Moore's law is applied to the number of cores on a single die, *i.e.* manufacturers increment the number of cores having about the same clock cycle.

Existing dusty-deck code is not capable of delivering increased performance with these new technologies, *i.e.* it is not able to exploit the parallelism offered by multi-core processors.

The exploitation of the improvements offered by a multi-core processor can be obtained operating on two different sides.

*Implicit parallelism* is exploited transparently, *i.e.* without user intervention, with both hardware and software techniques. Hardware techniques for exploiting parallelism transparently include *pipelining*, *super-scalar execution* and *dynamic instruction reordering* while software techniques are actually oriented on the use of parallelizing compilers such as *Polaris* [22].

Actually the exploitation of implicit parallelism is limited to a moderate level of parallelism. In order to achieve higher degrees of parallelism the user identifies tasks which are able to execute in parallel. This exploitation pattern is known as *explicit parallelism.* It is possible to express explicit parallelism using both *shared memory* and *message passing* models.

The improvement of single-core processors made the existing sequential code to automatically exploit the new resources. *I.e.* a sequential algorithm written for a certain processor, when re-compiled for a processor two times faster, scales with a factor two without any modification.

Unfortunately the exploitation of multi-core resources is not "automatic" such as in the single-core case. Programming code which efficiently exploits multi-cores require significant programming efforts. Furthermore the effort is multiplied when someone wants to write code parametric with respect to the parallelism degree.

In this thesis we target this problem proposing `m`$^2$`df` (Multi-threaded Macro Data Flow), a multi-threading high level framework based on *Macro Data Flow* (MDF) technique, as a viable solution.

## 1.1 m$^2$df in a nutshell

This framework allows the user to express the computations in the form of graphs. The user can then instantiate multiple times these graphs and submit instances for the execution in a streamed fashion.

When an instance is submitted, the tasks composing it are distributed to a set of workers which carry out the computation, adapting to the number of available cores.

As said before the parallelism is exploited through the multi-threading technique, using the shared memory as a communication mechanism. These concepts will be deepened in the next Sections.

The framework is assumed to be the target back-end of the compilation process of high level parallel programs.

## 1.2 Multi-threading

With the massive advent of shared memory multi-core architectures the multi-threaded programming paradigm has become a viable alternative for the exploitation of distributed on-chip architectures.

### 1.2.1 Processes vs threads

With the term process we identify a program which is able to be executed together with other programs, and to co-operate with them. While a program is a passive set of instructions, a process is an execution instance of the instructions representing the program.

Several processes are allowed to run at the same time. Multiprogramming allows multiple processes to share processors and other system resources. The simultaneity of the execution can be simulated (in this case we speak about concurrency) or effective (in this case we speak about parallelism). [1]

Each CPU executes a single process at a time, however multitasking allows each processor to switch between tasks ready to execution. The switch between processes in execution happens frequently enough to give the user the perception of multiple processes running at the same time.

The switch operation is triggered and implemented differently, depending on the operating system.

Each process owns a set of resources, including:

**Instructions** an image of the executable, representing the program being executed;

**Memory** includes executable code, process specific data (such as global variables), a call stack to keep trace of active routines and a heap to support "dynamic allocation" of memory space;

**Descriptors** describe the system resources allocated to the process, such as files or sockets;

**Other** security attributes (*e.g.* permissions) and process context, consisting in register images, physical memory addressing supporting structures (*Relocation Table*) and so on and so forth.

Usually, a process consists of a single thread of execution but, depending on the operating system, it may be made up of multiple threads of execution which execute different parts of the process' instructions concurrently.

A thread of execution, briefly called thread, is defined as an independent stream of instructions that can be scheduled by an operating system. It is the smallest unit of scheduling.

While processes are not able to share resources, multiple threads within the same process can share instructions (*i.e.* the process' code), resources and the context of the process. Moreover the thread handling is generally cheaper than the process' one: thread creation and context switch is typically faster with respect to the case of processes, due to the sharing of resources.

Maintaining multiple control flows is accomplished because each thread owns a private

- Register set;
- Stack pointer;
- Scheduling properties;
- Program counter.

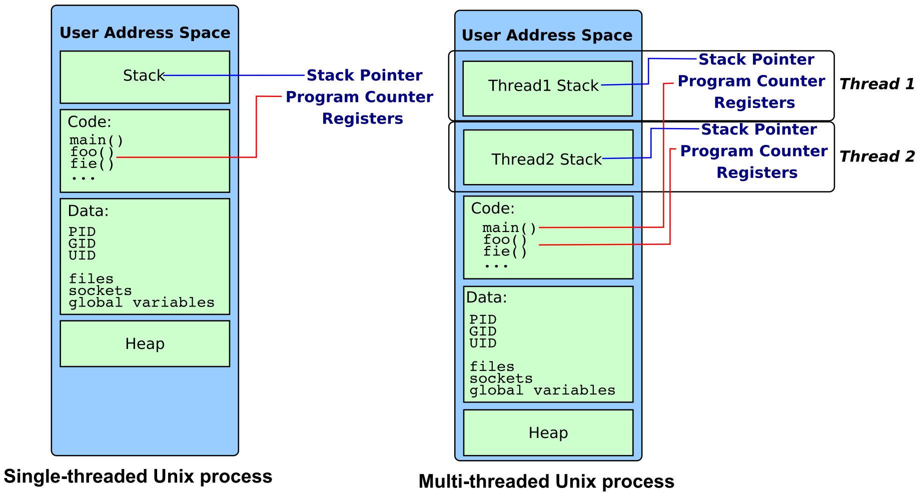


Figure 1.1: Comparison of a single-threaded and a multi-threaded process.

Figure 1.1 shows the comparison between a single-threaded Unix process and a multi-threaded Unix process.

Looking at the Figure we can easily notice that a thread exists within a process and uses the process resources. A thread represents an independent flow of control in the parent process, because of this a thread dies when the supporting process dies.

In order to exist, a thread duplicates the essential resources it needs in order to be independently schedulable.

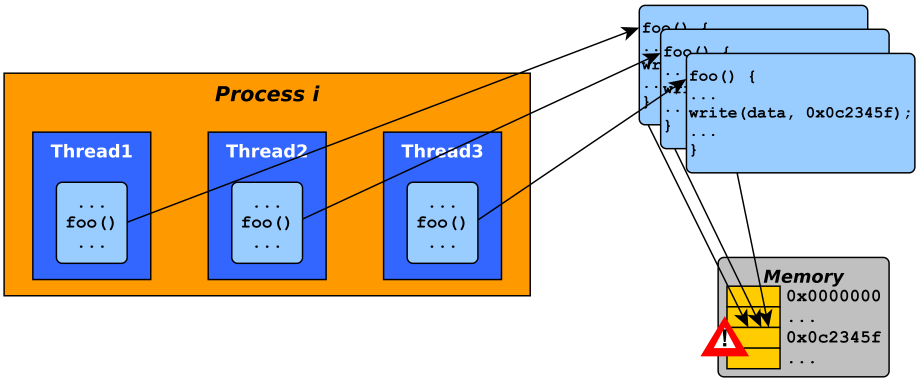
Threads share the process' resources, this fact imply that changes made by a thread to some shared resource will be seen by all other threads. The operations on shared resources require explicit synchronization, generally provided by the programmer.

### 1.2.2 Multi-threading pros and cons

Multi-threaded programming model provides developers with an abstraction of concurrent execution.

This programming model takes advantages to both single-CPU and multi-core systems.

In the first case a multi-threaded application is able to remain responsive to input. In a single-threaded program, if the execution blocks on a long-running task, the entire application can appear to freeze. This approach is heavily used by graphic libraries, which make the user interface support running on different threads than the business logic.



```
foo() {
...
write(data, 0x0c2345f);
...
}
```

Figure 1.2: Example of a race hazard caused by calls to external routines.

In the latter case, which is more interesting from our point of view, this programming model allows to work faster on multi-core and multi-processor (*i.e.* systems having more than one CPU) systems.

In fact the inter-process communication must be implemented through system mechanisms, and requires at least one memory operation. As threads in the same process share the same address space no intermediate copy is required, resulting in a faster communication mechanism.

On the other hand having multiple flows of execution operating on the same addressing space can lead to subtle issues.

Shared data may lead to race hazards: if two or more threads simultaneously try to update the same data structure they will find the data changing unexpectedly and becoming inconsistent. Bugs of this kind are often difficult to reproduce and isolate.

To prevent this kind of problems, multi-threading libraries often provide synchronization primitives such as mutexes and locks. These primitives allow the calling thread to atomically and exclusively operate on the shared data. Obviously a careless usage of these primitives may lead to other problems, such as deadlocks.

Another problem when working with threads relates to third party functions. If multiple threads need to call an external library routine, it must be guaranteed to be thread-safe (*i.e.* not leading to deadlocks or inconsistent data sets). If the routine is not thread-safe calls to that routines should be serialized. Figure 1.2 shows this situation.

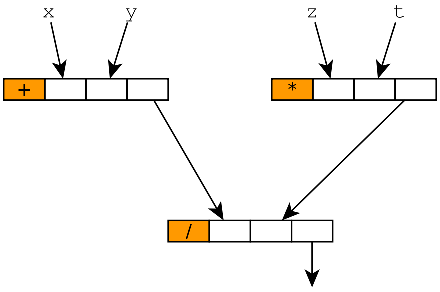


Figure 1.3: Data flow graph for the expression (x+y)/(z*t)

## 1.3 Macro Data Flow

The data flow (DF) computational model is a purely functional model of computation, born as an alternative to the imperative one [2]. According to this model executable programs are represented as graphs, called data flow graphs, rather than as linear sequences of instructions, as shown in Figure 1.3.

In a data flow graph, nodes correspond to instructions, and edges correspond to data dependencies between instructions. An instruction is enabled to the execution if and only if all of its input values, called *tokens*, are available. Once the instruction has been executed the input tokens are removed, and result tokens are produced and delivered the nodes next in the graph.

In this approach the instruction, instead of the process, is the unit of parallelism. In the DF model the ordering of the instructions is guaranteed by the data dependencies, hence it can be obtained through the application of the Bernstein's conditions.

The data flow model was not adopted because of some inherent limitations:

1. too fine grain parallelism: instruction level parallelism caused instruction fetching and result delivery costs to overcome performance gains;
2. difficulty in exploiting memory hierarchies and registers;
3. asynchronous triggering of instructions.

In order to overcome some of the issues presented by the data flow model, in the late '90s Macro Data Flow (MDF) technique was introduced. Informally, this technique applies all the data flow main concepts with a coarser grain: MDF instructions consists in entire portions of user code, also called *tasks*. This property of MDF makes it implementable in software without requiring new architectures. Furthermore the execution of the single user instructions can exploit all hardware optimizations offered by current processors.

A software implementation of MDF can offer several benefits:

- can run on existing multi core processors;
- it is possible to implement a MDF run-time support with standard tools, so it doesn't require ad-hoc compilers and tools (*e.g.* MPI);
- a general purpose interpreter can be used to express skeleton-based computations;
- the use of coarse grain instructions enables efficient serial code to be used to implement the tasks's behaviour.

In a multi-core environment most of the problems that the MDF exposes in a distributed implementation are naturally solved. First, there are no problems related to the execution of the MDF instructions by remote interpreters, such as cross-compilation or data serialization. Second, it is not mandatory the use of the message passing technique, in a local environment shared memory mechanism is efficient and worthwhile.

## 1.4 Organization of the work

In Chapter 2 we will take a look to the related work, in Chapter 3 we will examine the tools upon which `m`$^2$`df` relies.

In Chapters 4 and 5 we will deepen the design and the implementation of the framework. In Chapter 6 we will examine the `m`$^2$`df` performances and, finally, in Chapter 7 we will take the conclusions of this thesis, exposing some possible future perspective.

# Chapter 2

# Related work

This chapter targets the principal research trends around data flow and multi-threading.

First, in Section 2.1 we will present the main industrial standard tools for thread programming. Then, in Section 2.2 we will discuss the two main models related to data flow present in the state of the art: *Scheduled Data Flow* and *Data Driven Multi-threading.* Finally, in Section 2.3 we will discuss the main limitations of the presented approaches.

## 2.1 Thread programming

Programming truly multi-threaded code often requires complex co-ordination of threads. This can easily introduce subtle bugs due to the interleaving and synchronization of different processing flows on shared data. Furthermore, the efficient programming of parallel systems is mostly performed at a low level of abstraction (*e.g.* POSIX threads) and requires a deep knowledge of the target system features and makes.

In order to help the programmer in this work some higher level libraries, such as *OpenMP* [23], and languages, such as *Cilk/Cilk++* [25] were developed.

These libraries provide a programming model based on shared memory and work-sharing: new threads are forked for performing the work in parallel and then they are joint together when the parallel section is completed. This sequence of operations happens for each parallel section occurs in the program.

Both OpenMP and Cilk leave to the user the handling of thread synchronization providing proper constructs.

The languages and libraries discussed above are industrial products which have affirmed during the last years.

## 2.2 Data flow models

Many research trends are related to the data flow technique. Some of these look for optimizations of the data flow model, in order to overcome its limitations. Some other trends are oriented to the use of data flow extensions, such as Macro Data Flow, in order to exploit commodity hardware. Other trends explore the field of scheduling and optimization of graph-based computations according to different criteria.

### 2.2.1 Scheduled data flow

Scheduled Data Flow (SDF) [5, 6, 7] is an architectural approach which addresses some of the limitations of the pure data flow model. SDF aims to provide high performance decoupling instructions execution from memory accesses.

In this architecture a thread is a set of SDF instructions associated with a data frame and a state, called *Continuation*, present in memory. At thread level the system behaviour is the data flow one.

On the other hand the execution engine, also called *Execution Pipeline*, relies on control flow-like sequencing of instructions (*i.e.* it relies on a program counter). Instructions are fetched by an instruction fetch unit. The correct ordering of instruction is guaranteed by compile-time analysis. Execution Pipeline executes the instructions of a thread using only registers.

The decoupling of execution and memory access is performed by the *Synchronization Pipeline*. This component of the architecture is in charge of load the data of a thread from the memory to a register set before the execution of a thread (pre-load operation) and store the results of the execution from the register set to the memory after the execution of a thread (post-store operation).

According to the configuration of its continuation, a thread can be in one of four possible states: *Waiting Continuation* (WTC), *Pre-load Continuation* (PLC), *Enabled Continuation* (EXC), or *Post-store Continuation* (PSC).

A special unit, called *Scheduling Unit*, handles the management of continuations. Figure 2.1 shows the life cycle of a thread among the functional units.

Both Synchronization and Execution Pipeline must be able to communicate with local memory, registers and control logic in one clock cycle. This is a reasonable assumption as long as the number of pipelines in a chip remains low. As the number of pipelines grows the communication times between architectural components grows, becoming a limiting factor for the system

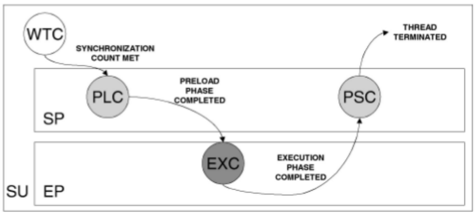


Figure 2.1: Life cycle of a thread in SDF. *Figure taken from [5].*

scalability.

In order to make the SDF architecture more scalable, the concept of clustering resources has been added [8]. In this architecture each cluster has the same architecture. A cluster consists of a set of processing elements and a *Distributed Scheduler Element*, which is in charge of balance the work among the processing elements.

### 2.2.2 Data-driven multi-threading and TFlux

Data Driven Multi-threading (DDM) is a non-blocking multi-threading model [9]. In this model a thread is scheduled for execution in a data flow fashion, thus eliminating the synchronization latencies.

The scheduling of threads is performed by an off-chip memory-mapped module, called *Thread Synchronization Unit* (TSU). TSU communicates with the CPU through the system bus.

The main difference between this approach and the SDF one consists in the fact that DDM doesn't require a special design processor, on the contrary it is studied to work with commodity microprocessors.

The core of this model consists in the TSU. This unit is in charge of handle synchronization between threads and schedule ready threads for execution.

Scheduling of threads is done dynamically at run-time according to data availability or to some cache management policy.

TSU is made out of three units. The *Thread Issue Unit* (TIU), the *Post Processing Unit* (PPU) and the *Network Interface Unit* (NIU). In addition the TSU contains a Graph Memory and a Synchronization Memory.

The TIU is in charge of scheduling ready threads. The queue containing the ready threads is split in two: the Waiting Queue and the Firing Queue. The first one contains the thread identification number (Thread#) of ready threads that are waiting for the prefetch of their own data from memory. Once the data is gathered the Thread# is shifted into the Firing Queue.

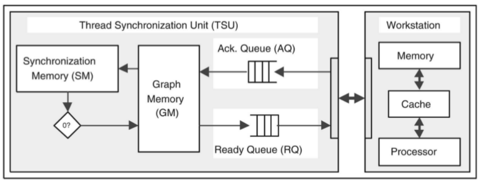


Figure 2.2: Thread Synchronization Unit structure and interaction. *Figure taken from [9].*

When the computation processor completes a thread, it passes the information about the completed thread to the PPU through the Acknowledgement Queue. The PPU uses the Thread# of the completed thread to index an associative memory, called Graph Memory, and get the Thread# of completed thread's consumers. The ready count of each consumer thread is decremented and, if it becomes zero (*i.e.* the thread becomes ready), the corresponding Thread# is forwarded to the TIU.
If a ready thread belongs to a remote node, its Thread# is forwarded to the NIU.

The last unit, the NIU, is responsible for handling the communications between the TSU and the interconnection network.
Figure 2.2 shows the overall picture of the TSU structure.

The DDM approach has been used to implement the *Thread Flux System* (TFlux). TFlux is a complete system implementation, from the hardware to the programming tools. The TFlux design is shown in Figure 2.3.

In order to make TFlux running on top of existing hardware and Operating Systems virtualization techniques have been used. First, the entire support is implemented at user level; second, the TSU functionalities are accessed through calls to high level library functions; third, the identification of the thread's bodies is done by the user through preprocessor directives.

These solutions allow TFlux to run on commodity Operating Systems, abstracting from the TSU implementation technique and permit an easy porting of existing C code to the TFlux system.

Abstracting from the TSU implementation technique led to different TFlux implementations. *TFlux Hard* implements the TSU using a dedicated hardware device, as in the DDM case. *TFlux Cell* is an optimized implementation for the IBM-Cell processor, in which the TSU functionalities are executed by the PPC element of the processor. At last *TFlux Soft* implements TSU as a software module running on a dedicated core of the target architecture.

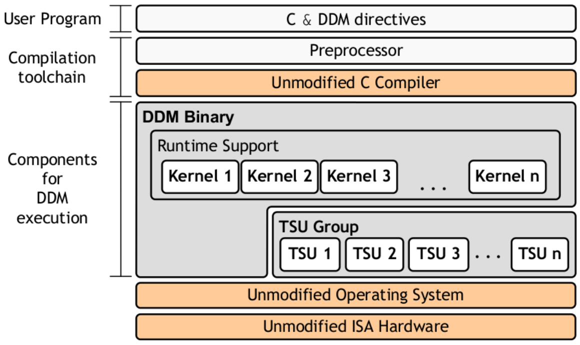


Figure 2.3: TFlux layered design. *Figure taken from [12].*

## 2.3 Limitations of the approaches

The approaches we discussed in this chapter have some limiting factors.

SDF is an hardware design approach, as such it requires a special design processor. This causes limitations to the usability of the architecture. A usable system should rely on off-the-shelf hardware and commodity Operating Systems.

DDM partially overcomes the SDF's limitations. It doesn't require a special processor, but requires additional custom hardware for implementing the synchronization functionalities.

Again, TFlux abstracts from the TSU implementation technique making its functionalities implementable in software. However, the implementation of the TSU functionalities requires a dedicated core to run.

In a MDF interpreter the scheduling entity is often idle (specially for coarser grain computations). It arises when tasks are completed for executing pre and post-processing phases. These facts imply a bad handling of the cores pool.

In this thesis we aim to overcome this limitation by implementing a different MDF-based system in which the scheduling entity arises just when needed, leaving for the rest of the time the entire cores pool to the computing entities.

# Chapter 3

# Support tools

In order to implement a portable system, `m`$^2$`df` was developed in such a way only standard libraries and mechanisms were used.

This chapter discusses the main tools on which `m`$^2$`df` relies for its functioning.

First, in Section 3.1, we will introduce the POSIX standard specifying in Sections 3.1.1, 3.1.2 and 3.1.3 the functions utilized in `m`$^2$`df`.

Then, in Section 3.2 we will introduce the system-dependent calls we have used deepening, in Section 3.2.1 the pipe mechanism.

## 3.1 The POSIX standard

POSIX (Portable Operating System Interface for uniX) is the name of a family of standards specified by the IEEE. The standard defines the Application Programming Interface (API), utilities and interfaces for software compatible with the Unix operating system family. The standard is currently supported by the majority of existing operating systems, although not Unix.

The last version of the standard, POSIX:2008, consists of three groups of specifications concerning:

- POSIX Base Definitions;
- System Interfaces and Headers;
- Commands and Utilities.

Depending on the degree of compliance with the POSIX standard, the operating systems can be classified on *fully* or *partly* POSIX-compliant. Between the fully POSIX-compliant systems one can mention Mac OS X, Solaris, LynxOS, MINIX and many others.

Other systems, while not officially certified, conform in large part including Linux, FreeBSD, NetBSD and many others.

Under Windows different solutions are possible, depending on the version. In Windows Vista and Windows 7 the UNIX subsystem is built-in, while in previous versions it must be explicitly installed.

The API functions specified by the standard are implemented in the POSIX C Library. This is a language-independent library, which uses C calling conventions. POSIX functionalities are often implemented relying on the C standard library.

These functionalities may be used by including several header files and target file system interaction, inter process communication, time handling and thread handling.

m$^2$`df` uses some POSIX facilities, mainly regarding thread handling, file descriptors control and system information gathering.

These functions are defined in the `fcntl.h`, `pthread.h` and `unistd.h` header files.

### 3.1.1 File descriptors control

In order to guarantee correctness, m$^2$`df` implementation needs the possibility of checking whether a pipe is full or is empty. The pipe mechanism, as we will see in Section 3.2.1, provides the a file abstraction and therefore we can use all the operations allowed on a file descriptor.

#### Flags control

The header file `fcntl.h` defines three function prototypes and a set of symbolic constants to be used by these functions.

In order to have non-blocking communications between interpreter threads and scheduler thread the function

```
int fcntl(
    int     filedes,
    int     cmd,
    ...
)
```

has been used. This function takes as input a file descriptor, *i.e.* the *filedes* parameter, a command, *i.e.* the *cmd* parameter and additional parameters related to the specified command.

It is possible to have a pipe with non-blocking behaviour by suitably setting the flags related to the file descriptor. This operation occurs in three steps: first, we need to get the value of the flags setted for the considered file descriptor; second, we modify the value of these flags by setting the non-blocking flag and finally we submit the new flag value.

For performing step one and step three we need to use the `fcntl` function. In particular the `F_GETFL` command tells `fcntl` to get the actual flags value and `F_SETFL` command tells `fcntl` to the new flags value.

### File descriptors polling

The header file `poll.h` declares the *poll* routine and defines several structures used by this routine.

```
int poll(
    struct pollfd   fds[],
    nfds_t          nfds,
    int             timeout
)
```

The `poll` routine examines a set of file descriptors contained in the *fds* array checking if some of them is ready for a set of events. The input array contains *nfds* entries. The *timeout* parameter specifies the maximum interval to wait for any file descriptor to become ready, in milliseconds. Relating to the value of the *timeout* parameter the function behaves differently: if *timeout* is `INFTIM` (-1), poll will block indefinitely, while if poll is zero the function will return without blocking.

This system call returns an integer value indicating the number of descriptors that are ready for I/O. If the timeout occurs with no descriptors ready `poll` will return zero. If some error occurs it will return -1.

In case of successful completion the occurred events will be stored, for each file descriptor, in the appropriate position of the *fds* array. In case of error, the *fds* array will be unmodified.

The routine's input array is composed of *pollfd* data structures. `poll.h` declares this structure as follows

```
struct pollfd {
    int     fd;
    short   events;
    short   revents;
};
```

The *fd* field contains the file descriptor to poll, the *events* field contains the events to poll for. The first two are input fields while the *revents* field is an output field. It will eventually contain the occurred events.

In order to simplify the events handling the same header file declares a set of bitmasks describing the events.

### 3.1.2 Thread handling

POSIX specifies a set of interfaces (functions, header files) for threaded programming commonly known as `POSIX threads`, or `Pthreads`. The access to the `Pthreads` functionalities occurs through the header file `pthread.h`.

In this file types are defined as well as constants and the prototypes of around 100 functions, all prefixed `pthread_` and divided in four major groups:

**Thread management functions** handling the life cycle of a thread and for handling thread attributes;

**Mutexes functions** to support the mutual exclusion (mutex) mechanism usage. These functions permit creating, destroying, locking and unlocking mutexes;

**Condition variables** functions concerning communications between threads that share a mutex, based upon programmer specified conditions;

**Synchronize** functions to manage read/write locks and barriers.

$\mathtt{m}^2\mathtt{df}$ heavily relies on Pthreads functionalities for creating and handling the multi-threading.

#### Thread creation

Once the scheduler or an interpreter is launched, by calling the `run()` method, a new thread is created. The creation phase consists in a call to the

```
int pthread_create
(
    pthread_t              *thread,
    const pthread_attr_t   *attr,
    void *(*start_routine)(void *),
    void                   *arg
)
```

function. Quoting the Pthread manual

> The pthread_create() function is used to create a new thread, with attributes specified by *attr*, within a process. If attr is NULL, the default attributes are used. If the attributes specified by attr are modified later, the thread's attributes are not affected. Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by *thread*.

The new thread will execute the function pointed by *start_routine* argument with arguments pointed by *arg* parameter.

### Thread pinning

POSIX models different cores of a single multi-core chip as processors, therefore in this text the terms core and CPU will assumed to be interchangeable.

CPU affinity is the ability of enforcing the binding of single threads to subsets of the available cores or processors. This operation is very important in order to get reliable results on actual multi-core chips.

CPU affinity can be divided in two main types:

**Soft affinity** also called *natural affinity*, is the capacity of the scheduler to keep processes/threads on the same core as long as possible. This is an attempt: if the scheduling of a certain process on the same core is infeasible, it will migrate on another core, resulting in the so called ping-pong effect;

**Hard affinity** is a requirement, provided through system calls, that enforces the scheduling of specific processes/threads to specific cores.

Why is CPU affinity so important? Three main benefits can be achieved.

The first one is that CPU affinity allows to exploit cache hierarchies. Binding a thread on a single core simplifies cache management and the data needed by that thread can be easily maintained on the cache of a single processor.

A second benefit is related to the first one: pinning multiple threads that access the same data to the same core will make these threads not contending over data, avoiding or reducing cache invalidation overhead.

The last benefit is not related to caches. Pinning the thread to a single core will avoid the ping-pong effect, resulting in a saving of clock cycles due to the reduced scheduler activity.

In order to provide hard CPU affinity the POSIX library provide a set of function and macros, different for processes and threads. Thread affinity can be set through the function

```
int pthread_setaffinity_np(
    pthread_t       thread,
    size_t          cpusetsize,
    const cpu_set_t *cpuset
)
```

Concerning this function, the Pthread manual says

> The `pthread_setaffinity_np`() function sets the CPU affinity mask of the thread *thread* to the CPU set pointed to by *cpuset*. If the call is successful, and the thread is not currently running on one of the CPUs in cpuset, then it is migrated to one of those CPUs.

Type *cpu_set_t* is a data structure representing a set of CPUs. This data type is implemented as a bitset. cpu_set_t is treated as an opaque object.[1]. All manipulations of this structure should be done via the set of macros defined in the `sched.h` header file.

Only a few of these macros are useful for our purposes

```
void CPU_ZERO(
    cpu_set_t    *set
)

void CPU_SET(
    int          cpu,
    cpu_set_t    *set
)

int CPU_ISSET(
    int          cpu,
    cpu_set_t    *set
)
```

The use of these macros is quite intuitive. CPU_ZERO takes as input a pointer to a cpu_set_t and clears it, so it contains no CPUs. CPU_SET adds the CPU identified by *cpu* parameter to the set pointed by *set.* CPU_ISSET simply tests whether the set pointed by *set* contains the cpu identified by *cpu*, this macro returns 0 if cpu is not contained in set and a non-zero value otherwise.

CPUs are identified by integer numbers between 0 and $N_{CPUs} - 1$.

In `m`$^2$`df` the scheduling of each interpreter is enforced to a single processor identified by its id number. Id numbers are integers between 0 and $N_{interpreters} - 1$.

If the support is launched with a number of interpreters greater then the number of available CPUs, all the supernumerary interpreters are allowed to run on each available CPU.

The scheduler thread is allowed to run on any available CPU, for efficiency reasons. In this way any time an interpreter thread stops because it has no work to do, the scheduler runs on the free core and schedules other ready tasks according to the scheduling policy.

### Thread termination

There are several ways in which a Pthread may be terminated. First, the thread may return from its starting routine; second, it may call the pthread_exit function; third, the thread can be terminated by other threads

---

[1] Opaque objects are allocated and deallocated by calls that are specific to each object type. Size and shape of these objects are not visible to the user, who accesses these objects via handles. More information at `http://www.mpi-forum.org/docs/mpi-11-html/node13.html`

through pthread_cancel or pthread_kill functions and fourth the entire process can terminate due to `SIGKILL` signals or to a call to `exec` or `exit` routines.

In `m`$^2$`df` the thread termination occurs differently for interpreters threads and scheduler thread. The first ones terminate by receiving a kill signal from the scheduler, while the second one terminates by calling the exit function.

```
void pthread_exit(
    void      *retval
)

int pthread_cancel(
    pthread_t    thread
)
```

As the Pthread man page says

> The `pthread_exit`() function terminates the calling thread and returns a value via *retval* that (if the thread is joinable) is available to another thread in the same process that calls pthread_join. [...] When a thread terminates, process-shared resources (e.g., mutexes, condition variables, semaphores, and file descriptors) are not released. [...] After the last thread in a process terminates, the process terminates as by calling exit with an exit status of zero; thus, process-shared resources are released.

The `pthread_cancel` routine requests to the thread indicated by *thread* to be cancelled. The cancellation operation takes place according to the target thread cancellation state. The target thread controls how quickly it terminates, according to its cancellation state.

The cancelation state consists in *cancelability state* and *cancelability type.* Cancelability state can be set to `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE` while cancelability type can be set to `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

Cancelability state determines whether a thread can receive a cancelation request or not. Cancelability type determines when a thread can receive cancelation requests. A thread with deferred cancelability can receive requests only at determined cancelation points, while asynchronous cancelability means that the thread can receive requests at any time.

By default a thread has cancelation state *enabled* and type *deferred.* These values can be set through the `pthread_setcancelstate` and `pthread_setcanceltype` routines.

In `m`$^2$`df` the scheduler cancels the interpreter threads, in this way interpreters must not check for termination messages.

The `pthread_exit` description is related to joinable threads. The join operation permits synchronization between threads.

```
int pthread_join(
    pthread_t   thread,
    void        **retval
)
```

The `pthread_join` function makes the calling thread to wait for *thread* to complete. *retval* is a placeholder for the *thread* exit status passed when calling `pthread_exit`.

The life cycle of the threads in m$^2$df will be targeted in the next chapter.

#### Inter-thread synchronization

The pthread library provides several mechanisms to suggest synchronization between threads. The main ones are mutexes and condition variables.

m$^2$df uses both of these mechanisms. Mutexes have been used in order to have exclusive and atomic access to the shared communication queues and conditions have been used for the implementation of semaphores.

**Pthread mutexes** Mutexes can be used for preventing race conditions. Typically the action performed by a thread owning a mutex consists in the updating of shared variables. Using a mutex represents a safe way of updating shared variables.

Mutex stands for mutual exclusion. A mutex acts like a lock protecting the access to a shared resource.

The basic concept of a mutex is that only one thread at a time can lock a mutex. Even if multiple threads try to lock the same mutex, only one will succeed. Other threads will actively have to wait until the owning one unlocks the mutex.

When several threads compete for a mutex, the losers will block at the lock call. Pthread also provides a non-blocking call, the trylock operation.

Mutexes in pthread must have the type `pthread_mutex_t`. A mutex must be initialized before it can be used. When initialized a mutex is unlocked. The initialization operation can be performed statically, through a macro, or dynamically through the function

```
int pthread_mutex_init(
    pthread_mutex_t                 *mutex,
    const pthread_mutexattr_t       *attr
)
```

This function initializes the mutex referenced by *mutex*. The *attr* object is used to establish properties for the mutex object, and must be of type `pthread_mutexattr_t`. If attr is `NULL`, a default attribute will be used. Multiple initializations of the same mutex result in an indefinite behaviour.

It is possible to free a no longer needed mutex through the function

```
int pthread_mutex_destroy(
    pthread_mutex_t      *mutex
)
```

This function destroys the object referenced by *mutex*. It is safe to destroy an unlocked mutex; trying to destroy a locked mutex results on undefined behaviour. A destroyed mutex becomes uninitialized, this object can be re-initialized using pthread_mutex_init.

The possible operations on a mutex are

```
int pthread_mutex_lock(
    pthread_mutex_t      *mutex
)

int pthread_mutex_trylock(
    pthread_mutex_t      *mutex
)

int pthread_mutex_unlock(
    pthread_mutex_t      *mutex
)
```

All these operation operate on the object referenced by *mutex*.

The mutex is locked by calling `pthread_mutex_lock`. If the mutex was already locked the calling thread blocks until the mutex become available.

The `pthread_mutex_trylock` routine behaves exactly as pthread_mutex_lock except that if the mutex was already locked (by any thread, including the calling one) the call return immediately with an appropriate error.

A mutex can be unlocked using the `pthread_mutex_unlock` function. This function releases the referenced mutex. When this routine is called the mutex becomes available again. If some thread is suspended on the mutex the scheduling policy will be used in order to determine which thread shall acquire the mutex.

**Pthread conditions** The second synchronization mechanism provided by the pthread library consists in the condition variables.

While mutexes implement synchronization by controlling the access of the threads to the shared data, condition variables allow the threads to synchronize basing on the actual value of data.
Without mutexes, the programmer should make threads to continuously check if the condition is met. This busy waiting is very resource consuming, since the processor cannot be used by other threads.

A condition variable achieves the same result without busy waiting.

The initialization and destroying of a condition variable occurs similarly to the mutex one.

```
int pthread_cond_init(
    pthread_cond_t              *cond,
    const pthread_condattr_t    *attr
)

int pthread_cond_destroy(
    pthread_cond_t  *cond
)
```

The *pthread_cond_init* routine initializes the variable referenced by *cond*, with the attribute *attr*. If attr is NULL a default attribute will be used. Attempting to initialize an already initialized condition variable results on undefined behaviour.

The *pthread_cond_destroy* function deletes the condition variable passed through reference. Attempting to destroy a condition variable upon which other threads are currently blocked results on undefined behaviour.

All the operations allowed on a condition variable are performed in conjunction with a mutex lock. These operations consist in

```
int pthread_cond_wait(
    pthread_cond_t      *cond,
    pthread_mutex_t     *mutex
)

int pthread_cond_signal(
    pthread_cond_t  *cond
)

int pthread_cond_broadcast(
    pthread_cond_t  *cond
)
```

The *pthread_cond_wait* function blocks the calling thread until the condition referenced by *cond* is signalled. This function should be called when the mutex referenced by *mutex* is locked. This function automatically releases the mutex while it waits. When a signal is received the mutex will be automatically locked, so the programmer is in charge of unlocking the mutex when the thread terminates the critical section.

To wake up a single thread which is waiting on a condition variable the *pthread_cond_signal* routine is used. Instead, the *pthread_cond_broadcadst* unlocks all threads blocked in the condition variable *cond*.

### 3.1.3 Miscellaneous functions

POSIX allows an application to test either at compile or run-time the value of certain options and constants.

At compile-time this operation can be done by including `unistd.h` or `limits.h` and testing the values with the macros therein defined.

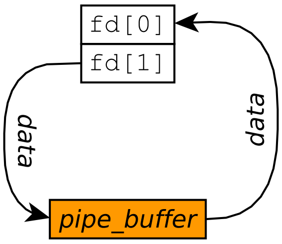


Figure 3.1: Working scheme of a pipe.

At run-time the programmers can ask for numerical values using the function

```
long sysconf(
    int        name
)
```

the return value meaning will depending on the request explicated by *name*. Anyway the values returned from this function are guaranteed not to change during the lifetime of a process.

In m$^2$df this function has been used to get the number of available cores using as command the constant `_SC_NPROCESSORS_ONLN`.

## 3.2 System dependent calls

The system dependent calls in m$^2$df are limited to the pipes communication system. For the sake of portability a version based on user-level communication mechanisms have also been implemented.

At compile-time is it possible to choose whether rely on the more efficient pipes mechanism or on the more portable pthread-based communication mechanism.

### 3.2.1 The pipe communication mechanism

Pipes provide a unidirectional interprocess communication channel. From the user viewpoint a pipe is a pair of file descriptors connected in such a way that the data written to the write end of the pipe will be available in the read end of the same pipe.

The file descriptors composing a pipe are not related to a real file. They are related to a buffer created and handled by the operative system kernel in main memory.

A pipe has a limited capacity. The capacity of the pipe is implementation-dependent, application should not rely on a particular capacity.

Read and write operation are performed through the functions used for reading and writing on a file, since the pipe mechanism provides a file abstraction.

A write operation on a full pipe will cause the writing thread to block until all the data have been written on the pipe. Similarly, a read operation will cause the calling thread to block until some data will be available on the pipe.

Read and write operation may be made non-blocking by setting the `O_NONBLOCK` flag on the read end or on the write end respectively. This flag can be set or clear independently for each end.How to set an operation non-blocking has been shown in Section 3.1.1.

In case the `O_NONBLOCK` flag have been set writing on a full pipe or reading on a empty pipe will fail.

A pipe is created using

```
int pipe(
    int       filedes[2]
)
```

This function creates a pair of file descriptors, pointing to the pipe buffer, and places them in the array *filedes*. `filedes[0]` will represent the read end and `filedes[1]` will represent the write end.

POSIX standard defines the parameter `PIPE_BUF`. This parameter specifies the number of bytes that can be atomically written into a pipe. The value of this parameter is implementation-dependent, but the standard guarantees to be at least 512 bytes.

Since threads in the same process share the file descriptors defined for the process, the pipe mechanism, can be used for inter thread communications even if it was originally designed to support inter process communications. Threads can exchange informations through a pipe which will efficiently synchronize them.

# Chapter 4

# Logical design

In this chapter we will discuss the design and the structure of the support this thesis proposes.

In Section 4.1 we will examine the global structure of the system and then, in Section 4.2 we will discuss the design of the m$^2$`df` system modules.

## 4.1 Overall picture

Figure 4.1 shows the overall picture of the m$^2$`df` system.

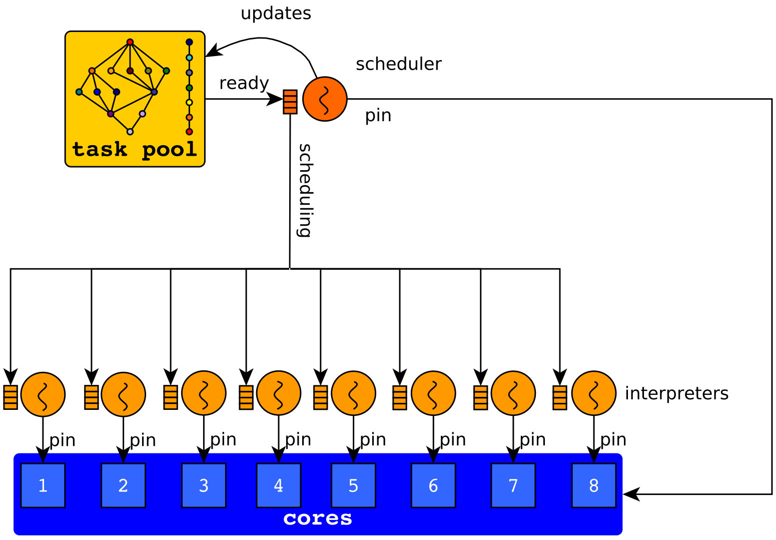


Figure 4.1: Overall picture of the m$^2$`df` system.

Several entities co-operate in order to guarantee the system working correctly. Especially, we can notice a set of interpreter threads and a single scheduler thread.

Each interpreter is pinned to (*i.e.* it is forced to run on) a specific available core while the scheduler thread can run on any available core.

Each interpreter thread simply executes the tasks delivered to it and signals their completion to the scheduler thread. If the tasks queue is empty the interpreter will suspend its execution letting the scheduler thread arise.

The scheduler thread is the once operating on the task pool. It handles the post-completion phase of the tasks by updating the ready counters of the successor tasks of a completed one, and pushing the ready tasks (*i.e.* the tasks whose ready counter has become zero) into a queue.

This thread then distribute the ready tasks among the interpreters, according to a scheduling policy.

It is possible, for the user, to submit different instances of the same graph for the execution with different input data sets. The task pool shown in the Figure contains instances instead of graphs. An instance is a graph copy associated with some input data.

The user thread invoking `m`$^2$`df` functionalities is allowed to run concurrently with the other threads until it invokes the `m`$^2$`df`'s termination function. A call to this routine will cause the calling thread to suspend its execution until the support terminates its execution.

This feature allows the user to operate in a streamed fashion, by asynchronously submitting new instances for the execution, and to wait for their completion safely.

## 4.2 Design of the support

`m`$^2$`df` provides functionalities for handling the MDF graph creation and instantiation and functionalities relative to the management of the computation. These functionalities co-operate in order to have an efficient and easy-to-use MDF computation.

In the following pages we will deeply describe the design of the modules composing `m`$^2$`df`, exploring the motivations that led to the actual design.

### 4.2.1 Graph management

The first class of functionalities provides the user with abstractions needed to operate on the internal implementation of the graph.

In `m`$^2$`df` there are two representations of the graph representing the computation: an abstract representation and a concrete representation.

The first one regards the aspects necessary for handling the dependencies between nodes of the graph. Hence the abstract representation keeps track of the ready count of each node and of its consumers.

The second one regards the aspects strictly related to the computation. This representation will keep track of the code to be executed, of the input data needed and of the locations where to put the results after the computation has completed.

These representations have been implemented through two data structures. The abstract representation keeps track of the concrete representation in order to speedy the scheduling operations.

The abstract representation contains information needed by the scheduler for handling the task pool, such as the consumer tasks or the ready count. The interpreters don't mind about these informations.

On the other hand, the concrete representation contains the informations strictly needed by the interpreters to execute the task, *i.e.* the code, the input data and the results placeholders.

Anyway the user doesn't access these structures. All the operations on the graph are performed through the wrapper class `Graph`.

This class exposes all the functionalities needed to operate on the graph, *e.g.* to create it by adding nodes and edges and to submit new instances for the execution.

As the user builds the graph, the class creates an internal representation of the computation. This representation separately keeps track of the graph structure and of the routines to be executed by each node of the graph.

The graph structure consists in an array of `abstract_node` structs. Each abstract node refers to its consumers as a list of indexes in the array.

When the user submits a new instance for the execution a new concrete representation of the graph is created. This representation is ready to the execution.

The concrete representation consists in a copy of the abstract one in which each node points to a `task` structure. The task is created and correctly initialized during the instantiation operation.

Each task contains the code to be executed, an array of input data and an array of pointers used to put the results in the correct places. In more detail the results array contains pointers to the correct locations (according to with the graph structure) in the input buffers of the consumer tasks. The set-up of the pointers is performed during the instantiation.

Each instance of the graph, represented by the `Instance` class, can run only one time. An instance manages a concrete graph, this class allows two operations: *1)* the starting of the execution, through the `start` method and *2)* the management of the completion of a task.

The first operation consists in pushing the first task of the graph into the ready queue (*i.e.* enabling it to the execution).

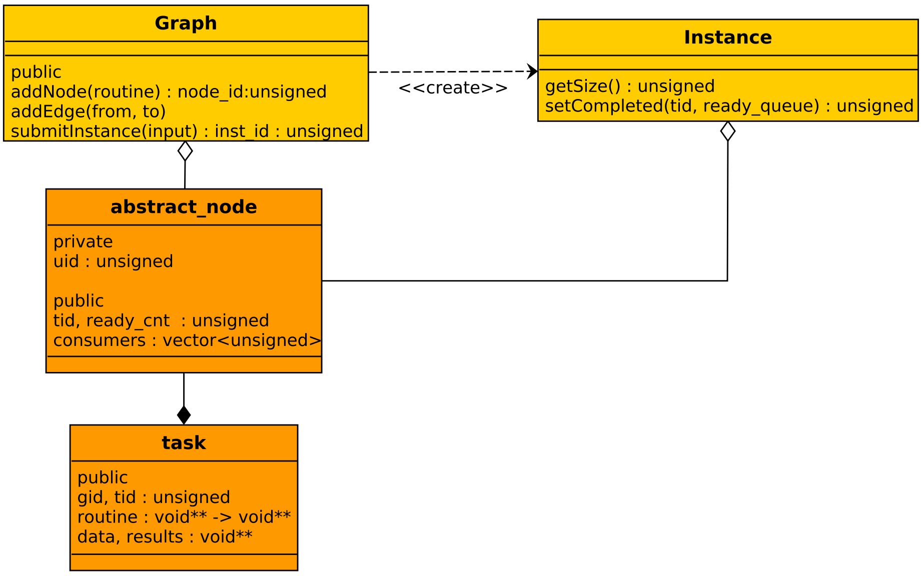


Figure 4.2: Class diagram of the graph management.

The second operation consists in decrementing the ready counters of all the consumers, and pushing them in the ready queue whether their counter becomes zero.

Figure 4.2 shows the class diagram related to this part of the support.

The functionalities related to the graph management are exposed to the user through the Graph class. The user creates a graph through the methods `addNode` and `addEdge` and submits new instances through the method `submitInstance`.

At present we manually built the testing graphs but, from the perspective of future work, the user should be represented by a compiler, eventually driven by user directives (*e.g.* `#pragma`) indicating the tasks and their interconnections.

Submitting a new instance will cause the creation of an object of the class Instance. This object will contain a concrete representation of the graph. The constructor of the Instance class will ensure that the concrete representation is correct.

Once the new instance has been created, it is passed to the scheduler which will begin the execution by putting the first task in the ready queue, through the `start` routine.

When a task is completed the scheduler will call the `setCompleted` method. This method will decrement the ready counter of all the consumers of the completed task. If the ready counter of some task will reach zero this method will enable them for execution, by pushing them in the ready queue. The return value of this method indicates the number of tasks which have been enabled. In order to reduce latencies due to the fetching of data from memory when a task becomes ready its input data is pre-fetched.

Actually, the pre-fetching is supported only for the `gcc` toolchain through the `__builtin_prefetch` function. Quoting the `gcc` docs:

> This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions will be generated. If the pre-fetch is done early enough before the access then the data will be in the cache by the time it is accessed.

For the sake of portability the calls to `__builtin_prefetch` should be substituted by inlined segments of assembler code, implementing the pre-fetching of data. Modern architectures provide assembler annotations which make it simple to implement.

### 4.2.2 Data management

For the sake of portability `m`$^2$`df` was conceived to work with tasks respecting the pattern `void**`⇝`void**`.

Dealing with raw memory pointers is often difficult, especially for inexpert C/C++ programmers, and can lead to memory corruptions and segmentation faults.

In order to simplify the data management, *i.e.* input and output from a task, some utility classes were designed. Figure 4.3 shows the class diagram of these classes.

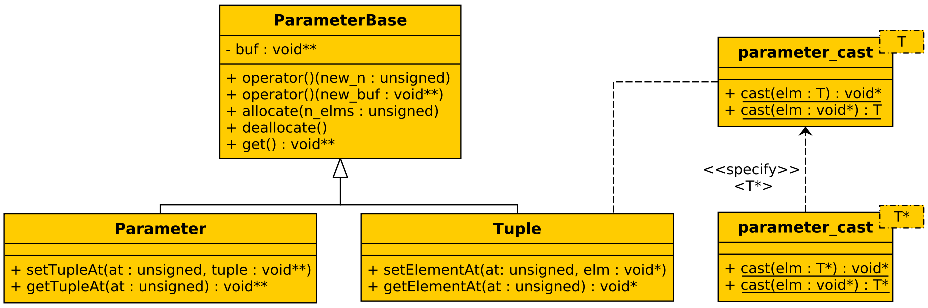


Figure 4.3: Class diagram of the data management support.

The hierarchy shown in the Figure above consists in three classes and

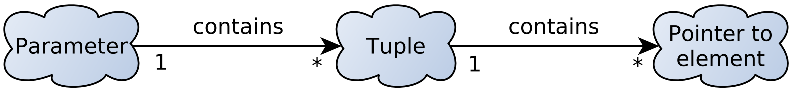


Figure 4.4: Organization of data.

two container classes.

The class `ParameterBase` implements the basic methods needed to handle the raw memory chunk. These methods gives the user the possibility to implicitly or explicitly allocate a memory block and to deallocate it.

Implicit allocation is allowed by the `operator()` overload. It is possible to allocate a new memory segment through the `operator()(unsigned)` method or to handle a pre-existent one, through the `operator()(void**)`.

Explicit allocation is allowed only for new chunks. It is implemented by the `allocate` method.

In order to guarantee a correct memory handling the deallocation of the memory is leaved up to the user. The user can explicitly deallocate a memory chunk through the `deallocate` method.

The `get` method simply returns the row pointer. This pointer can be used to return the output from the task or to set a tuple into the parameter.

Two classes extend the previous one. The first one, the `Tuple` class, represents an heterogeneous set of elements. It is possible to set the elements into the tuple using the `setElementAt` method and to get them using the `getElementAt` method.

These methods operate with `void*` type. In order to have a simple and safe cast from and to this type two template classes have been implemented.

The cast operation is slightly different for simple types and pointer types. Because of this the `parameter_cast<T>` was thought to cast simple types while `parameter_cast<T*>` specifies its behaviour for pointers.

These classes have just two overloads of the same `cast` static method. The overloads implement the cast from and to `void*` type.

The second class is the `Parameter` class. This class represent an aggregation of tuples. Tuples can be set and get through the `setTupleAt` and `getTupleAt` methods.

This model, as shown in Figure 4.4, allows to easily create and manage parameters consisting of multiple and heterogeneous elements.

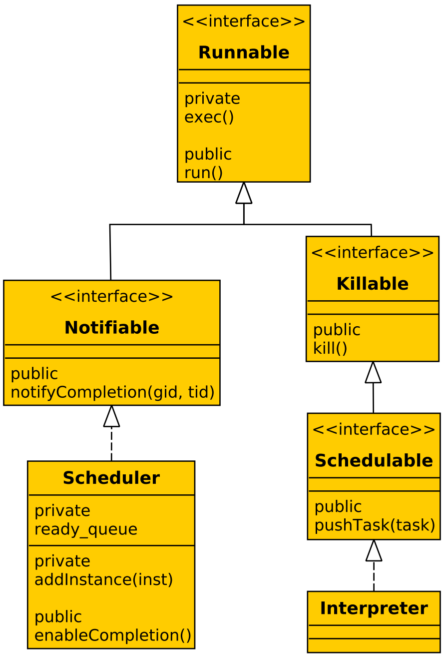


Figure 4.5: Class diagram of the MDF support.

### 4.2.3 Computation management

The remaining part of the design relates to the support to the computation. This part consists in a MDF run-time support.

The support is provided by a pool of classes describing the workers which execute the tasks and the master which distributes the tasks to be computed among the workers.

In the model shown in Figure 4.5 class `Interpreter` represents the behaviour of the workers above, while class `Scheduler` represents the behaviour of the master.

The classes Scheduler and Interpreter are strictly coupled, since they have deep interactions. The scheduler needs to send the tasks to be executed to the interpreter and to receive an acknowledgement when a task is completed. These communications are inherently asynchronous.

In order to reduce the coupling between these classes we designed a hierarchy of interfaces.

Both scheduler and interpreters are `Runnable` classes. As such they must implement two methods: `exec` and `run`. The first is a private one, it implements the behaviour of the class. The second instead has the aim of creating a new thread to execute the behaviour coded by exec method.

Subsequently the interfaces specify the functionalities required from the single class.

As far as the interpreter is concerned, the scheduler is a `Notifiable` object. Interpreter can notify the completion of the task <gid, tid> through the method `notifyCompletion`.

The scheduler perceives the interpreter as a `Schedulable` object. The scheduler can submit a new task for the execution through the method `pushTask` and, once terminated the execution, can kill the interpreter through the `kill` method.

The computation is entirely managed by the run-time support. The user isn't aware of these interactions, he/she interacts with the support through some interface functions.

It is possible to start the support through the `start` function, to specify the number of interpreters needed through the `tune` function and to wait for the end of the computation through the `finalize` function.

If the user doesn't explicitly specify the number of interpreters the support automatically use one interpreter for each available core in the target architecture.

### 4.2.4 Communications management

All the communication concerns have been grouped and handled by the `shared_queue` class. This class implements a thread-safe queue. Each interpreter has a shared queue by which receives the tasks to be executed. The scheduler has a shared queue by which it receives the notifications of the task completion.

These queues are private fields of the classes, and the access occurs through appropriate methods.

Figure 4.6 shows the design of this part of the support. The class `shared_queue` implements a thread-safe FIFO queue. All the operations on the queue are guaranteed to be atomic and deadlocks free.

The `shared_queue` class provides methods to push a new element, to pop the top element, to test whether the queue is full or empty and setting read and write operation as blocking or non-blocking.

For the sake of portability this class has been designed as a generic one, its implementation must not rely on the type of data contained by the queue.

In order to handle the blocking communications and to keep track of the empty or full queue cases the `Semaphore` class have been used. This class implements the functions of a semaphore relying on the pthread synchronization primitives.

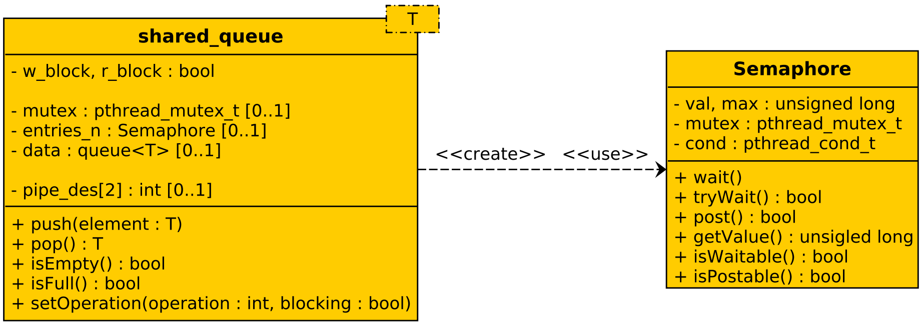


Figure 4.6: Class diagram of the communications support.

The Semaphore supports the wait and post operations. In addition to these operations Semaphore allows the tryWait operation, which performs the wait operation iff this operation doesn't block the calling thread, and testing whether the semaphore is safely waitable or postable.

Also the operations on the Semaphore are guaranteed to be atomic, however avoiding deadlocks is in charge to the user.

### 4.2.5 Global overview

Figure 4.7 shows the overall picture of the classes used to implement the system.

In particular it is possible to point out the interactions between different classes.

The scheduler and the interpreters communicate through different shared queues: each interpreter receives the tasks to execute into a private queue and sends the completed tasks's ids into a scheduler's private queue. Appropriate methods mediate the access to these queues.

The function `schedule` is in charge of performing the scheduling of tasks. The scheduler calls this function when it needs to schedule the ready tasks.

This function groups all the scheduling issues. When we need to modify the scheduling policy it is sufficient to change the behaviour of this function.

The diagram in Figure 4.7 shows another pure function, which was not discussed before. The function `thunk` is a pure function which takes as parameter a pointer to a Runnable class and uses it for calling the `Runnable::exec` method. In order to perform this method call the function thunk is a friend of the Runnable interface.

Indeed, it is needed to run the exec behaviour only after the forking of the supporting thread has been performed. The `pthread_create` function, called by the run method, receives as parameter the thunk body.

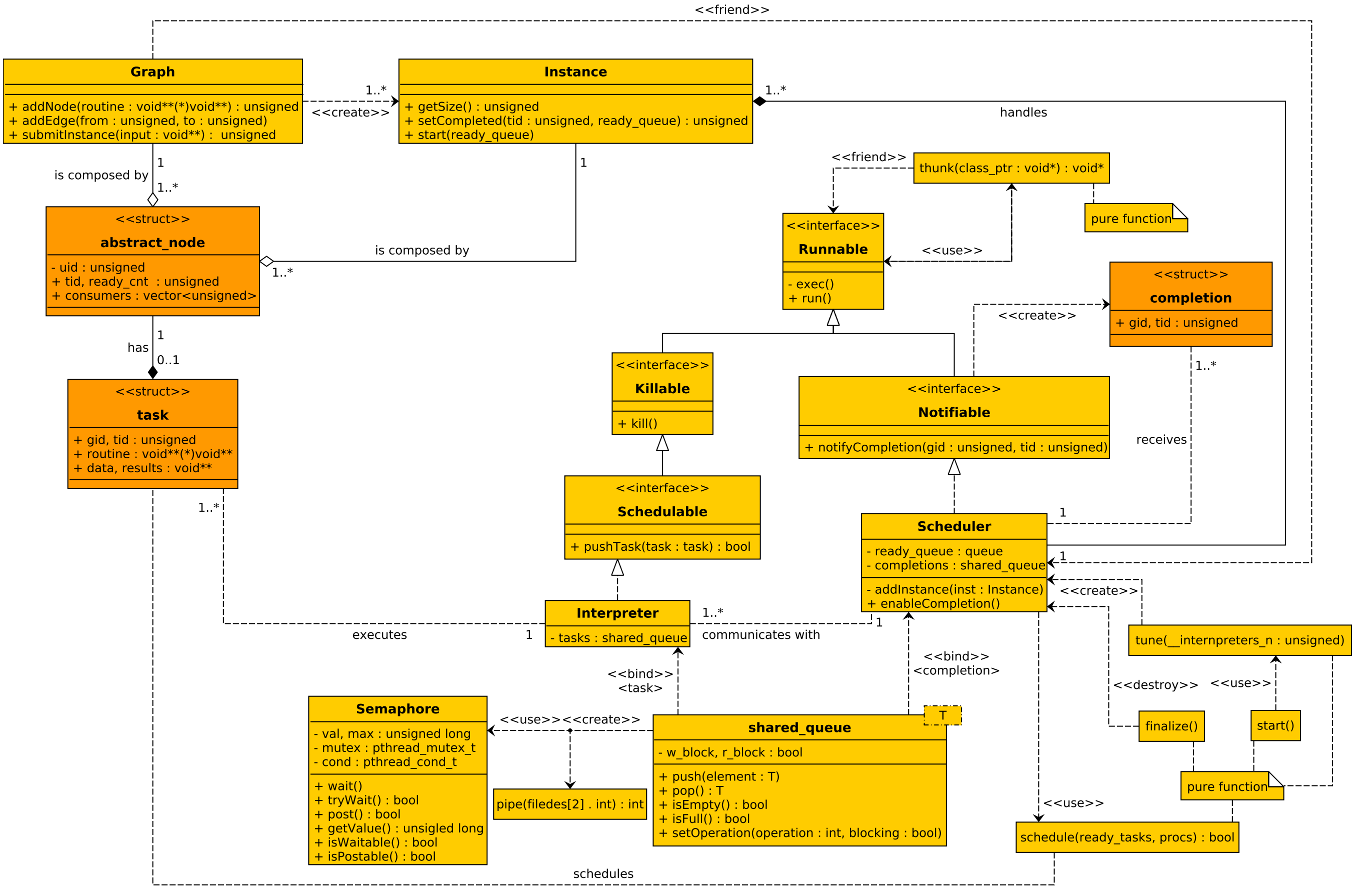


Figure 4.7: Global picture of the m$^2$df's system.

The interpreters notify a task completion by calling the `notifyCompletion` method. This method creates a new `completion` structure, which simply aggregates a graph id and a task id in a unique object, and pushes it into the queue of the scheduler.

Each interpreter maintains a reference to the scheduler in order to notify the completion in a simple way. This design also allows multiple levels of scheduling in which hierarchical schedulers distribute the tasks at their disposal to a set of subordinate interpreters.

The UML class diagram also shows that the Graph class is friend of the Scheduler one. This allows the Graph class to safely submit a new instance to the scheduler.

The method for adding a new instance (*i.e.* `Scheduler::addInstance`) has been set private since it represents a critical operation which implies synchronizations, deep knowledge of the scheduler's structure and some checks on the graph.

Keeping that method private enforces the user to submit new instances only through the `Graph::submitInstance` method. This method makes the appropriate controls on the graph to be submitted and submits in an efficient way the new instance to the graph (*i.e.* limiting at the most the synchronizations between threads).

## 4.3 Life cycle

`m`$^2$`df` is a multithreaded MDF run-time support. In this section we will describe how the threads in `m`$^2$`df` interact in order to carry out their computations.

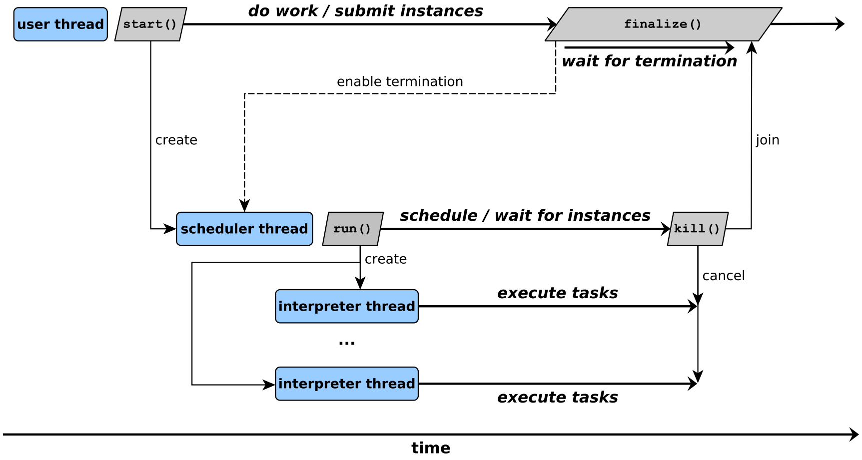


Figure 4.8: Life cicle and interaction of threads in `m`$^2$`df`.

Figure 4.8 shows the threads interactions. Initially, the application consists in a single thread, *i.e.* the user one. This thread builds the graphs and prepares the input data for the instances.

Once the user thread calls the `start` function, a new thread is created. This thread executes the scheduler function. The scheduler, when launched, creates a new thread for each interpreter and starts the delivering of tasks.

In the meanwhile the user thread is allowed to run, in parallel. It is possible to execute streamed computations, submitting new instances of the graph as soon as the input data is available, or to do other work. The user thread can wait for the end of the MDF computations by calling the `finalize` function.

A call to `finalize` will enable `m`$^2$`df` to terminate, as soon as it terminates to execute the submitted instances. Up to the point this function has not been called yet, the scheduler will wait for new instances when it terminates the scheduling of the submitted ones. A call to the finalize function will block the calling thread until `m`$^2$`df` has terminated its execution.

After a call to `finalize` a new computation can be started, with different configurations, by calling the function `start` once again.

Only one instance of the scheduler is allowed to run at a time. Multiple calls to the start routine, before finalize is called, will produce an exception.

# Chapter 5

# Implementative aspects

This chapter deals with the implementation concerns relative to the development of m$^2$`df`.

In Section 5.1 we will expose the general choices which influenced the entire project. In Section 5.2 we will discuss the implementation of the communication mechanism, specifying the two different versions we have implemented in Sections 5.2.1 and 5.2.2. Section 5.3 discusses the semaphore class' implementation.

In Section 5.4 we will treat the aspects related with the thread management, then in Section 5.5 we will examine the interpreter and scheduler behaviour, discussing in Section 5.6 the scheduling mechanism's implementation.

Finally in Section 5.7 we will enumerate the global variables we used in the project, describing their meaning.

## 5.1 General choices

In this section we will briefly explain some high level implementation choices related to the m$^2$`df`'s development.

First, the language chosen for developing this programming framework was C++. Indeed the decision of relying on the POSIX standard restricts the possible languages to C and C++.

C++ provides several benefits such as the possibility to use all object-oriented features, the templates mechanism, namespaces and all the functionalities and the containers of the C++ standard library maintaining performance comparable to that of the C language.

The second choice relates to the naming conventions. All the files composing m$^2$`df` are named following the pattern `mtdf_<component name>`. The user can access all the m$^2$`df`'s functionalities by including the header file `mtdf.h`.

In order to avoid naming collisions all the classes and routines composing `m`$^2$`df` have been grouped into the `m2df` namespace.

To simplify some interactions among threads the shared memory mechanism have been exploited through the use of global variables. The inner namespace `m2df::global` groups all the global variables, which meaning will be discussed in Section 5.7.

## 5.2 Communication implementation

The class `shared_queue` embeds all the communication concerns.[1] This class provides the abstraction of a FIFO queue, abstracting from the inner implementation details.

Actually two different communication mechanisms have been implemented. The fist one relies on the `pthread` synchronization primitives, while the second one relies on the `pipe` mechanism.

In order to have more performant communications the code implementing the `shared_queue` is chosen at compile time through the use of *conditional groups* `#ifdef ... #endif`.

It is possible to switch the communication mechanism by setting (or unsetting) the appropriate flag, defined in the `m2df_debug.h` header file.

The `shared_queue` class, is a *template* class. This allows implementing a generic shared queue, abstracting from the details related to the content of the queue. Listing 5.1 shows this chass' signature.

```
class shared_queue
{
    bool _w_block, _r_block;
    #ifdef M2DF_SYNC_VERSION
    pthread_mutex_t _mutex;
    Semaphore _entries_n;
    queue<T> _data;
    #endif
    #ifdef M2DF_PIPES_VERSION
    int _pipe_des[2];
    #endif
public:
    inline shared_queue();
    virtual ~shared_queue();

    void setOperation(short __op, bool __blocking);
    inline void push(T& __data);
```

---

[1] Because of design choices, as previously discussed in Chapter 4.

```
    inline T pop();
    inline T tryPop();
    inline T timedPop(unsigned time, bool *valid);
    bool isEmpty();
    bool isFull();
};
```

Listing 5.1: `shared_queue` class' signature.

### 5.2.1 Pthread-based mechanism

This version guarantees the FIFO ordering of the data through the `std::queue` class. This queue is maintained as a private field. The access to the queue is mediated through appropriate methods.

In addition to the queue, this class maintains a `pthread mutex` in order to guarantee the atomicity of the operations on the queue.

In order to provide blocking operations a `Semaphore` instance is maintained. The implementation details of the Semaphore class will be treated in Section 5.3.

Listing 5.1 shows the structure of the `shared_queue` class. When the framework is compiled according to this version, the preprocessor makes the class to have the structure defined through the `#ifdef M2DF_SYNC_VERSION ... #endif` guards.

Both the read and write operation can be set as blocking or non-blocking, the relative information is maintained by the `w_block` and `r_block` flags.

The implementation of the `push` and `pop` methods uses the `entries_n` semaphore is such a way that a thread can safely (*i.e.* without causing deadlocks) suspend its execution. Figure 5.1 shows the behaviour of these operations.

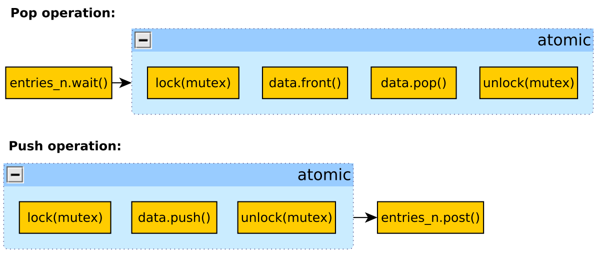


Figure 5.1: Push and pop operation on a queue with pthread primitives.

In case an of empty queue, the `wait` operation will cause the calling thread to block. Similarly, in case of a full queue, the `post` operation will cause the calling thread to block.

In case an operation is set as non-blocking and the considered operation cannot be performed (*e.g.* a push operation on a full queue or a pop operation on an empty queue) the invoked method will throw an appropriate exception.

Other operations allow to check whether the queue is full or empty. In this implementation these methods simply check if the associated `Semaphore` is postable or waitable, and return the result.

For the sake of extensibility different versions of the `pop` operation have been implemented. Precisely a non-blocking version, called `tryPop`, and a timed version, called `timedPop`.

These methods behave the same way if the queue is not empty: they return the top element, removint it from the queue. In case of empty queue the first one immediately returns false, while the second one waits up to *time*[2] milliseconds for the queue to become ready. If the timeout expires then it returns a `false` value, otherwise it returns a `true` value and the top element.

### 5.2.2 Pipe-based mechanism

In the version based on the pipe mechanism the FIFO ordering is guaranteed by the pipe itself. In this implementation the class only contains the file descriptors needed by the underlying mechanism. These descriptors figure through the `#ifdef M2DF_PIPES_VERSION ... #endif` guards, as shown in Listing 5.1.

In this case, setting an operation blocking or non-blocking is a little bit more complicated with respect to the other version. The `setOperation` method is in charge of setting the value of the appropriate field of the class and, in addition, it is in charge of setting the flags on the appropriate file descriptor through the `fcntl` routine.

The `push` and `pop` operation will simply invoke the `write` and `read` operations on the appropriate file descriptor.

As previously explained in Section 3.2.1 the POSIX standard provides the `PIPE_BUF` parameter specifying the number of bytes that can be *atomically* written to a pipe. The standard doesn't specify this parameter's value, but guarantees to be at least 512 bytes. In order to ensure the atomicity of these operations we decided to send only pointers to the tasks through the pipe[3].

The write operation will write a pointer to the object to push into the pipe,

---

[2] *time* is an input parameter of the method.

[3] This choice also allows to abstract from the `task`'s structure, increasing expandability.

and the read operation will read it from the pipe. The invoked operation will not return before it has completed.

In case an operation is set as non-blocking the invoked routine (*i.e.* `read` or `write`) will return an error in case the requested operation cannot be completed. The method will throw an appropriate exception in case it gets the above error.

The following listings show this behaviour.

```
void push(T& __data) {
    T* temp=&__data;
    if(write(pipe_des[1], &
        temp, sizeof(T*)) ==
        -1) { //writing error
        if(!w_block) throw
            FullQueueException
            (); //throw
            exception only if
            non-blocking

        cerr<<"Error␣in␣
            writing␣into␣pipe:
            ␣"<<strerror(errno
            )<<endl;
        cerr.flush();
        throw
            PipeWriteException
            ();
    }
}
```

```
T pop() {
    T *ret;
    if(read(pipe_des[0], &ret,
        sizeof(T*)) == -1) {
        //reading error
        if(!r_block) throw
            EmptyQueueException
            (); //throw
            exception only if
            non-blocking

        cerr<<"Error␣in␣
            reading␣from␣pipe:
            ␣"<<strerror(errno
            )<<endl;
        cerr.flush();
        throw
            PipeReadException
            ();
    }
    return (*ret);
}
```

In order to check if the queue is full or empty the class relies on the `poll` function. The `isEmpty` method polls the read-end of the pipe, checking if there is something to read. Similarly, the `isFull` method polls the write-end of the pipe, checking if it is possible to write something.

The poll method returns an integer which indicates several possible situations. These methods will convert this value to a boolean and return that value.

## 5.3 Semaphore implementation

As shown in Section 4.2.4 the Semaphore class provides methods for executing classical semaphore operations. The implementation of this class relies on some pthread synchronization primitives, mainly on pthread mutexes and conditions.

We choose to re-implement the semaphore functionalities from scratch, instead of using POSIX semaphores, because of some inherent problem with this API[4]. Furthermore we needed to easily and safely check whether the semaphore was waitable or postable, operation not permitted with the POSIX `semaphore`.

Listing 5.2 shows the declaration of the class. This class maintains a counter (*i.e.* the *val* field), an upper bound for this counter, a mutex and a condition.

```
class Semaphore {
    unsigned long val, max;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
public:
    /*Methods*/
};
```

Listing 5.2: Semaphore class declaration.

All the operations on the semaphore are guaranteed to be atomic. The mutex is utilized in order to implement the atomicity.

The suspension is performed using the pthread condition.

In case the user invokes a `wait` operation on a semaphore with val equal to zero the calling thread will wait the condition. The condition mechanism guarantees a correct blocking, without deadlocks. When the thread will be waked-up it will decrement the val counter and continue the execution. Again, the condition mechanism guarantees to wake-up a single thread in case more than one is waiting.

The `tryWait` operation will decrement the `val` counter if and only if this operation will not block the calling thread. Otherwise the method will return the `false` value.

On the contrary, the `post` operation is in charge to increment the `val` counter. This operation is correctly performed if val is lower than the upper bound set for the semaphore, otherwise it returns a negative response.

Having a value equal to one after the counter increment, means that some thread is suspended on the semaphore. In this case the method posts the condition variable in order to wake-up one of the waiting threads.

The remaining operations allow to test whether the semaphore is waitable or postable. These tests are performed in an atomic way too.

The first control returns true if the `val` counter is equal to zero. The second one returns true if the `val` counter is less than the upper bound.

---

[4] Mainly related to the `errno` variable.

The following listings show the implementation of wait and post operations.

```
void Semaphore::wait()
{
    pthread_mutex_lock(&mutex)
        ;
    if(val == 0) {
        suspended = true;
        pthread_cond_wait(&
            cond, &mutex);
        suspended = false;
    }
    val--;
    pthread_mutex_unlock(&
        mutex);
}
```

```
bool Semaphore::post()
{
    bool success = true;
    pthread_mutex_lock(&mutex)
        ;
    if(val < max) val++;
    else success = false;
    if(val == 1 && success ==
        true) {
        pthread_cond_signal(&
            cond);
    }
    pthread_mutex_unlock(&
        mutex);
    return success;
}
```

## 5.4 Thread creation and pinning

The `pthread_create` routine requires, as input, a start routine which must respect the template `void*(void*)`. In `m`$^2$`df` we need to create new threads executing methods related to classes.

In order to overcome this limitation we defined a `thunk` function. As shown in Listing 5.3 this function takes as an input parameter a pointer to the object on which invoke the method, casts the pointer to the `Runnable` interface and invokes the `exec()` method.

```
void *thunk(void *__class_ptr) {
    Runnable *instance = (Runnable*)__class_ptr;
    instance->exec();
    return NULL;
}
```

Listing 5.3: Thunk routine.

The `Runnable::run()` method is in charge of creating the new thread. This operation is shown in Listing 5.4, for the scheduler creation. The interpreter creation is similar, but no parameter attribute is passed to the `pthread_create` function.

```
struct sched_param my_param;
pthread_attr_t my_attr;
int error_code;

```

```
//set high scheduling priority
pthread_attr_init(&my_attr);
pthread_attr_setinheritsched(&my_attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&my_attr, SCHED_OTHER);
my_param.__sched_priority = sched_get_priority_max(SCHED_OTHER);
pthread_attr_setschedparam(&my_attr, &my_param);

if((error_code = pthread_create(&thread_info, &my_attr, thunk, (
    void*)this)) != 0)
    throw ThreadForkException(error_code);
```

Listing 5.4: Thread creation.

After the threads have been fired, in `m`$^2$`df` we need to set the CPU affinity of threads to subsets of the available cores.

The programmer specifies, through an appropriate parameter, the number of interpreters he wants to start. This value is maintained into the `global::processors_number` global variable.

This variable is set into the `tune` function, during the initialization phase. If the user doen't specify this value, `m`$^2$`df` will fire an interpreter for each available core in the target architecture. We retrieve the number of available core through the `sysconf` function, as described in Section **??**.

The interpreters are identified through an `id`. Ids are unsigned integers between 0 and $N_{int} - 1$. Each interpreter is allowed to run on the $i^{th}$ core, where $i$ is the interpreter's id.

In case the programmer launches some excess interpreters, namely $N_{int} > N_{cores}$, the excess interpreters are allowed to run on any available core. Listing 5.5 shows the pinning of the interpreter threads played by *Interpreter::run()* method.

```
cpu_set_t cpu_set;
unsigned cpu_num = global::processors_number;

CPU_ZERO(&cpu_set);
if(iid < cpu_num)
    CPU_SET(iid % cpu_num, &cpu_set);
else for(unsigned i = 0; i < cpu_num; i++)
        CPU_SET(i, &cpu_set); //excess parallelism interpreters
            can run in any cpu
if((error_code = pthread_setaffinity_np(thread_info, sizeof(
    cpu_set_t), &cpu_set)) != 0) //some error occurred
    throw ThreadSchedulingException(error_code);
```

Listing 5.5: Pinning of interpreter threads.

Similarly, the scheduler is allowed to run on any available core. We can permit this thread to run on any free core because its tasks are often quick and, in addition its intervention is fundamental to make the computation to proceed, since it schedules the new tasks to be calculated.

Furthermore the scheduler has higher scheduling priority to prevent it to be a performance bottleneck. Listing 5.6 shows part of the *Scheduler::run()* method.

```
unsigned cpu_number = MIN(global::processors_number, procs.size
    ());
cpu_set_t cpu_set;

CPU_ZERO(&cpu_set);
for(unsigned cpu_id = 0; cpu_id < cpu_number; cpu_id++)
    CPU_SET(cpu_id, &cpu_set); //scheduler can run in any core

if((error_code = pthread_setaffinity_np(thread_info, sizeof(
    cpu_set_t), &cpu_set)) != 0) //some error occurred
    throw ThreadSchedulingException(error_code);
```

Listing 5.6: Pinning of scheduler thread

## 5.5 Interpreter and Scheduler loops

Both interpreter and scheduler classes implement the `Runnable` interface. This interface declares a public method `run` and a private method `exec`.

The first method is in charge of creating the new thread and setting its properties, such as scheduling priority. This method also performs the pinning of the thread on one of the available cores.

Once the thread has been created the `exec` method implements the specific behaviour of the class.

The interpreter implementation of this method executes a loop in which:

- waits for a new task;
- executes the received tasks;
- stores the results in the consumer tasks;
- notifies the completion to the scheduler.

Listing 5.7 shows this behaviour. The execution is terminated by the scheduler which cancels the executing thread.

Looking at the code in Listing 5.7 we can observe that it *copies res into t.results*. That line means that the interpreter, once it has completed a task, copies the results tokens directly into the neighbour tasks' input buffers. In this way no intermediate copies are required and the scheduler's work amount is minimized.

```
while(true) { //interpreter is "killed" by the scheduler
    register task t = tasks.pop();

    void **res = t.code(t.data);

    copy res into t.results

    sched_addr->notifyCompletion(t.gid, t.tid);
}
```

Listing 5.7: Interpreter loop pseudo-code.

On the other hand the scheduler thread executes a different loop, shown in Listing 5.8. The first thing we can notice is that the scheduler loop contains some `mutex` operations. These operations are necessary in order to guarantee the user to operate in a streamed fashion: he can submit new instances to the execution, through the `Graph::submitInstance` method. This method is allowed to invoke the `Scheduler::addInstance` private method[5]. The concerned method loads the new tasks, and pushes the instance's root task into the ready queue, these operation must be performed in an atomic way with respect to the scheduler loop.

We can also see that the scheduler terminates its execution only when all tasks have been completed and it was enabled to complete. The enabling is performed through the `Scheduler::enableCompletion` method, which simply sets a flag. This method is invoked by the `finalize` function, so $\texttt{m}^2\texttt{df}$ cannot terminate before the `finalize` function has been called.

Once the loop is terminated, the scheduler cancels all the interpreter threads (which are all idle) and exits.

## 5.6 Scheduling

The scheduler instance performs the scheduling operation. This operation is performed by invoking the `schedule` routine.[6]

This function implements the scheduling policy: takes a `std::queue` of ready tasks and a `std::vector` of interpreters as input and returns a boolean value indicating whether a task was scheduled or not.

The scheduling is task-based. Actually the schedule function pops a ready task from the queue and sends it to one interpreter in a round-robin fashion.

---

[5] Since the `Graph` class were declared as `friend` of the `Scheduler` one, as described in Section 4.2.5.

[6] As we saw in Section 4.2.5

```
while(true) {
    register unsigned enabled_tasks = 0;

    if completion is enabled && all tasks have been completed:
        break

    while(schedule(ready_tasks, procs)) ; //schedule until there
         is something to schedule
    pthread_mutex_unlock(&mutex);
    //now it is possible to submit new instances
    register bool cond;
    do {
        completion compl_task = completed_tasks.pop();
        unsigned c_gid = compl_task.gid, c_tid = compl_task.tid;

        pthread_mutex_lock(&mutex);

        enabled_tasks = graphs[c_gid].setCompleted(c_tid,
            ready_tasks);
        compl_tasks++;

        if(enabled_tasks) {
            while(schedule(ready_tasks, procs)) ;
            enabled_tasks = 0;
        }

        cond = (compl_tasks < tasks_n);
        pthread_mutex_unlock(&mutex);
    } while(cond);
}

for(unsigned i = 0; i < procs.size(); i++) {
    procs[i]->kill(); //cancel interpreter threads
}

pthread_exit(&compl_tasks);
```

Listing 5.8: Scheduler loop pseudo-code.

In case changes to the scheduling policy are needed, it is sufficient to modify the behaviour of this function.

## 5.7 Global variables

In order to simplify the interactions between different parts of the support a set of global shared variables have been grouped into the `m2df::global` namespace.

These variables are setted once and accessed in a read-only fashion.

`processors_number` this variable represents the number of available core. It is set by the `tune` routine when starting the computation and is read by the scheduler and the interpreters for simplify the pinning operations. It is possible to receive this value from the user, *i.e.* as an input to the `tune` routine, or to automatically set it, *i.e.* getting it through the `sysconf` routine, as described in Section 3.1.3;

`scheduler_addr` this variable is a pointer to the scheduler instance. It is set by the start routine. The Graph class uses this variable in order to submit new instances to the scheduler, and the finalize routine uses this pointer for enabling the completion and joining the scheduler thread;

`comm_mutex` almost all classes use this mutex for printing debug messages having single access to the output buffer.

# Chapter 6

# Experiments

In this chapter we will discuss the experiments performed in order to evaluate $\texttt{m}^2\texttt{df}$ performance.

In Section 6.1 we will present the target architectures used for the tests, in Section 6.2 we will examine the benchmarks used and present the results and finally in Section 6.4 we will discuss the presented results.

## 6.1 Target architectures

The $\texttt{m}^2\texttt{df}$'s validation experiments have been performed on two machines:

- The first one, called `ottavinareale`, is a multiprocessor with two *Intel Xeon*® model *E5420*.

  This model relies on the Penryn architecture. It provides 32 KB L1-data cache, 32 KB L1-instruction cache dedicated to each core. The L2 cache is shared between cores of the same processor and it is sized 6 MB.

  The system runs a Linux kernel version 2.6.18-194.

- The second one, called `andromeda`, is a multiprocessor with two *Intel Xeon*® model *E5520* [28].

  This CPU is based on the Nehalem architecture. With this architecture Intel leaves the Multi-Chip Package approach for the monolithic one. All the cores composing the CPU are integrated on the same die, differently from the Core2-Quad which was composed by two Core2-Duo dies on the same base.

  The specific CPU model provides 4-cores, each of which has a dedicated 64 KB L1 Cache (further divided in 32 KB L1-data cache and 32 KB L1-instructions cache) and a dedicated 256 KB L2 cache. The chip also contains a shared 8 MB L3 cache. In this configuration the L2 cache behaves as a buffer with respect to the L3 cache.

Furthermore this model adopts the QuickPath Interconnect technology [29]. This technology allows any processor to efficiently access the data contained into other processors' caches in a NUMA fashion by-passing the system bus.

The Xeon processor also implements the *Hyper-Threading*$^{TM}$ technology. This is a Simultaneous Multi-Threading implementation which makes the Operative System "*to see*" a number of logical processors higher than the physical one. Specifically, this model has four 2-threaded cores.

The utilized model has a clock frequency of 2.26 GHz, with a *TurboBoost*$^{TM}$ of 2.53 GHz.

`andromeda` is a multi-processor with 8 physical cores and 16 logical cores running a Linux kernel 2.6.18-164. The experimental results bring out that the Hyper-Threading does not add further benefits with respect to the exploitation of the physical cores.

Some tests required the use of optimized linear algebra routines, mainly related to the BLAS standard API and the Lapack library.

As efficient implementation of these functionalities we targeted the libFLAME routines. libFLAME is an open-source C library with Lapack functionalities, developed and maintained by the University of Texas. For our tests we used libFLAME, version 3.0-5861.

All tests were built using compilers of the GCC suite, version 4.1.2.

## 6.2 Experiment setup

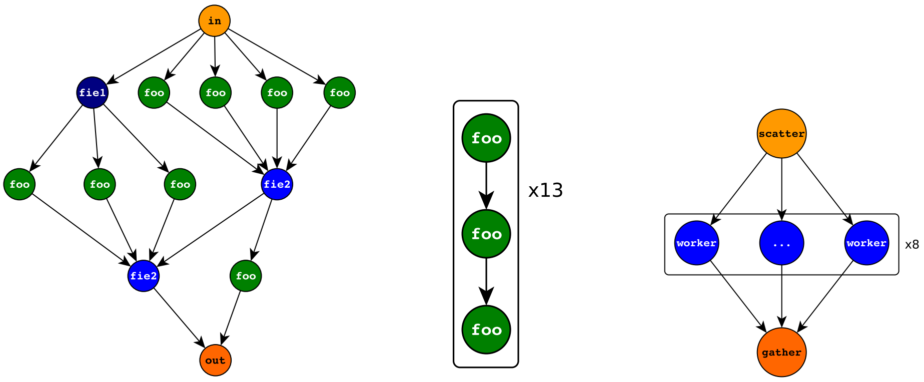
In this Section we will describe the tests performed on $\mathtt{m}^2\mathtt{df}$.

For each test we measured the completion times of the sequential implementation, namely $T_{seq}$, and of the parallel one, namely $T^{(N)}$, with parallelism degrees of $N = 2, 4, 8$.

In order to have more precise results we took as completion time the average time of five runs.

For each test we evaluated the speed-up as

$$s(N) = \frac{T_{seq}}{T^{(N)}}$$



(a) Generic (b) Pipeline (c) Map

Figure 6.1: Kind of graphs used for synthetic application tests.

and we extracted the efficiency as

$$\varepsilon(N) = \frac{\frac{T_{seq}}{N}}{T^{(N)}} = \frac{s(N)}{N}$$

Speed-up measures how much the parallel version of the computation improves with respect to the sequential implementation while the efficiency normalizes that measure with respect to the parallelism degree.

### Preliminary tests

The first tests we performed on `m`$^2$`df` mainly consisted in synthetic applications. These tests were performed in order to test the correct working of the run-time support and to study its behaviour with different grains.

For the sake of completeness we will describe these tests too, reporting their results.

For these tests we used a *generic graph*, *i.e.* a graph without a particular structure, a 13-stages *pipeline graph* and a *map graph* with 8 workers.

Figure 6.1 shows the structure of these graphs. Each node in the graph, but the input nodes, simply increment the value of a variable for a number $N$ of times.

Figures 6.2, 6.3 and 6.4 show the results of the preliminary tests performed on `ottavinareale` and on `andromeda` without variance on the task's work amount.

Tests have been performed also with a random variance up to 25% on the N value, *i.e.* $new\_N = N \pm rand()\%N \cdot 0.25$.

For these tests, which operate on a stream of 128 instances, we also measured the completion time the single instance as the average time on ten runs. In this way we can have in mind the order of magnitude of the elaboration time of the single stream element. The measures relative to this time are reported in Table 6.2.

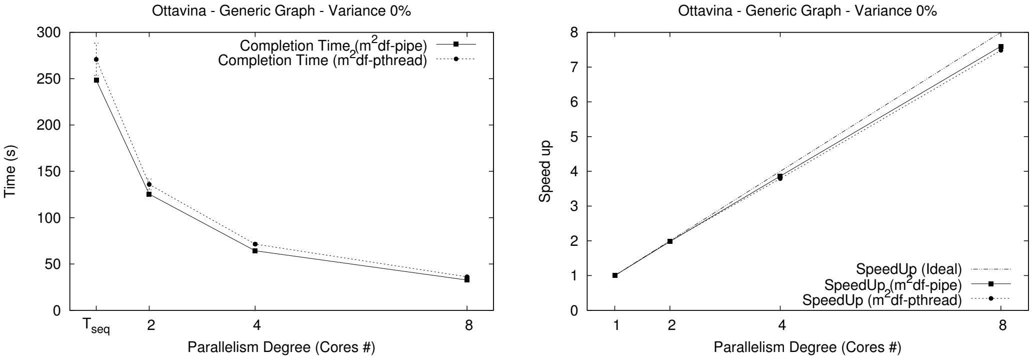
|  | `ottavinareale` | `andromeda` |
|---|---|---|
| Version 1 - 0% | $2.4189 \pm 0.0039$ | $2.8902 \pm 0.0015$ |
| Version 1 - 25% | $2.5457 \pm 0.2573$ | $2.8752 \pm 0.2931$ |
| Version 2 - 0% | $3.0766 \pm 0.0165$ | $3.1876 \pm 0.0031$ |
| Version 2 - 25% | $2.6473 \pm 0.8621$ | $3.1784 \pm 1.0349$ |
| Version 3 - 0% | $1.7045 \pm 0.005$ | $1.9614 \pm 0.0019$ |
| Version 3 - 25% | $1.6174 \pm 0.086$ | $1.9237 \pm 0.103$ |

Table 6.1: Single shot execution times for synthetic applications, *in seconds*.

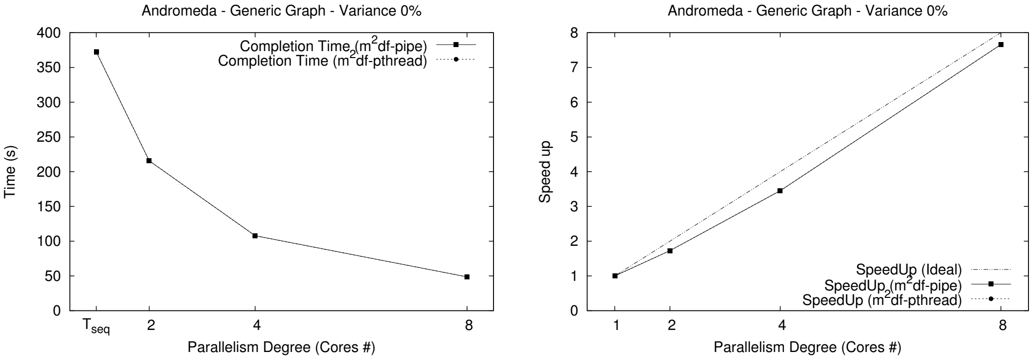
Figures 6.5, 6.6 and 6.7 show the results of these tests.

We can notice how the tests have a different behaviour in the two machines: while on `andromeda` the pipe-version and the pthread-version have the same trend, on `ottavinareale` the two versions have slightly different trends (Figures 6.2(a), 6.5(a), etc).

More in general we can notice how on `andromeda` speed up always follow the ideal linear trend, independently from the variance. On the other hand, on `ottavinareale`, speed up follow the linear trend with one of the two version, while the less performant is at worst at 80% of the ideal speed up.



(a) Completion time (b) Speed up[1]

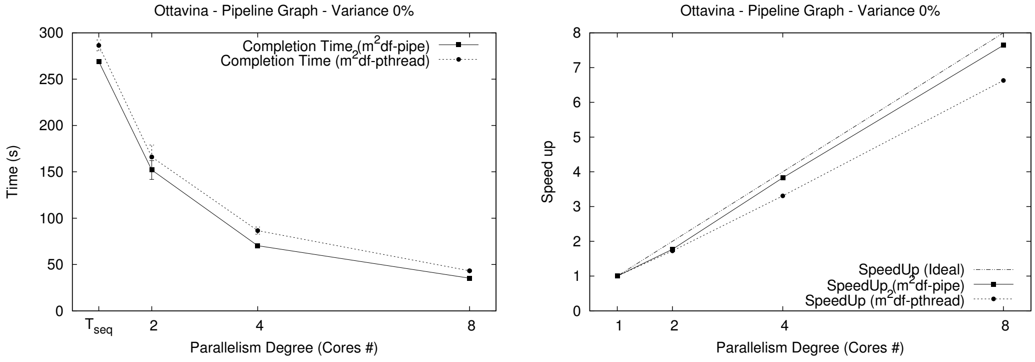


(c) Completion time (d) Speed up[1]

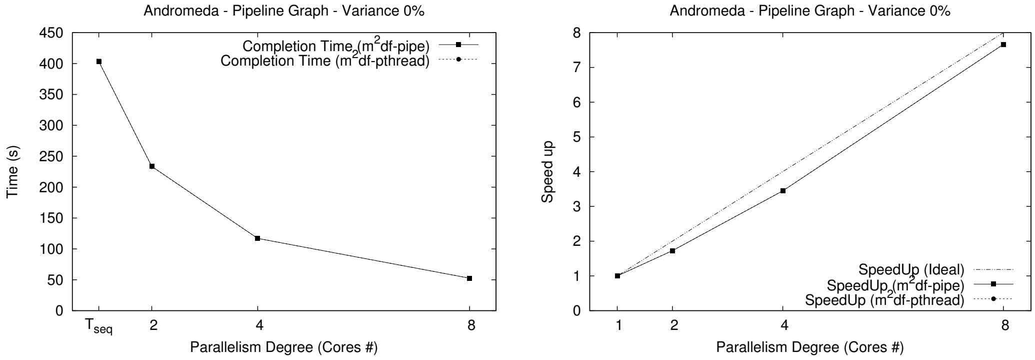
Figure 6.2: Results of Generic Graph computation, with variance of 0%.

---

[1] Computed as described in Section 6.2.

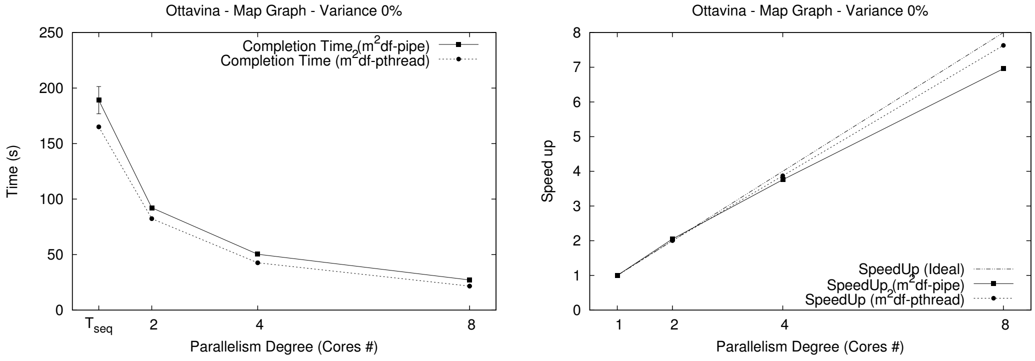


(a) Completion time (b) Speed up$^1$

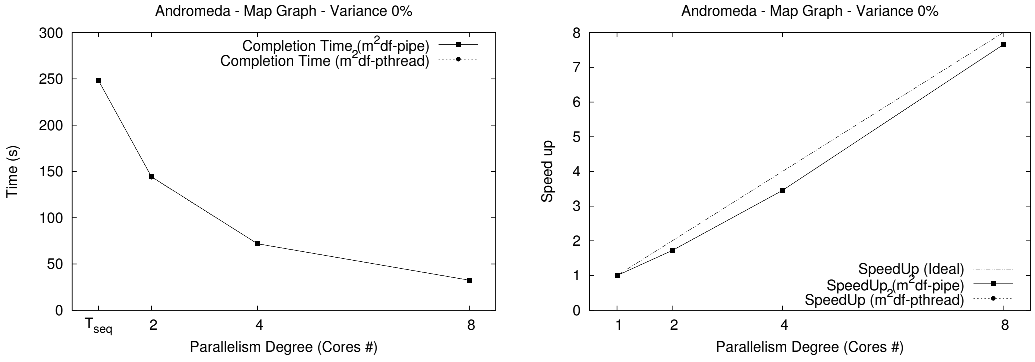


(c) Completion time (d) Speed up$^1$

Figure 6.3: Results of Pipeline Graph computation, with variance of 0%.

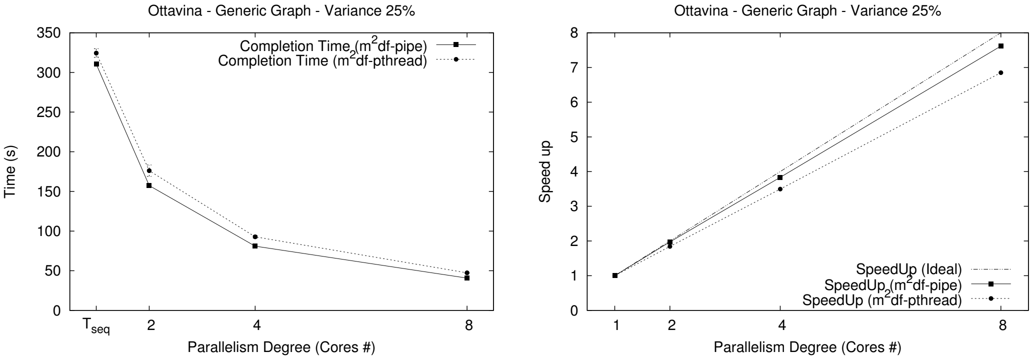


(a) Completion time (b) Speed up$^1$



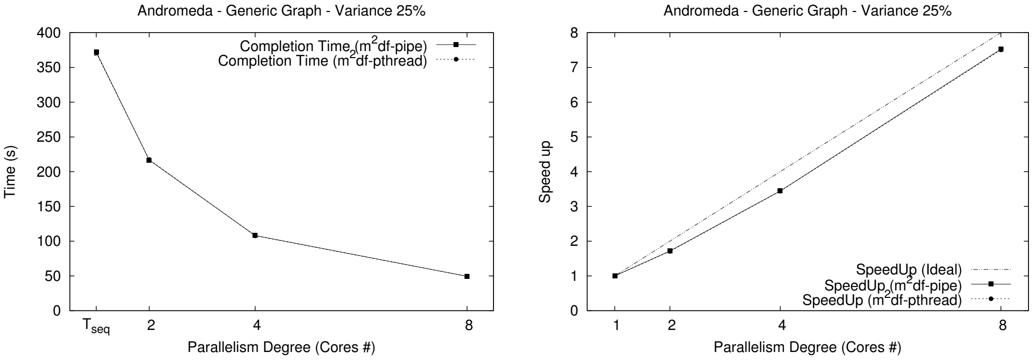
(c) Completion time (d) Speed up$^1$

Figure 6.4: Results of Map Graph computation, with variance of 0%.



(a) Completion time

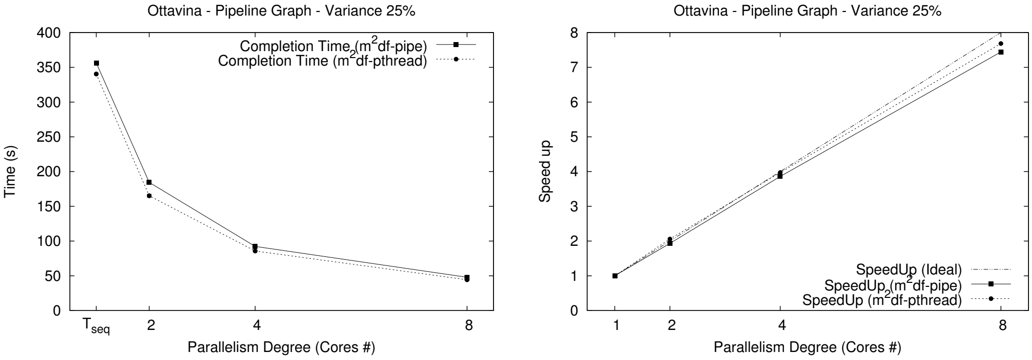
(b) Speed up[1]



(c) Completion time

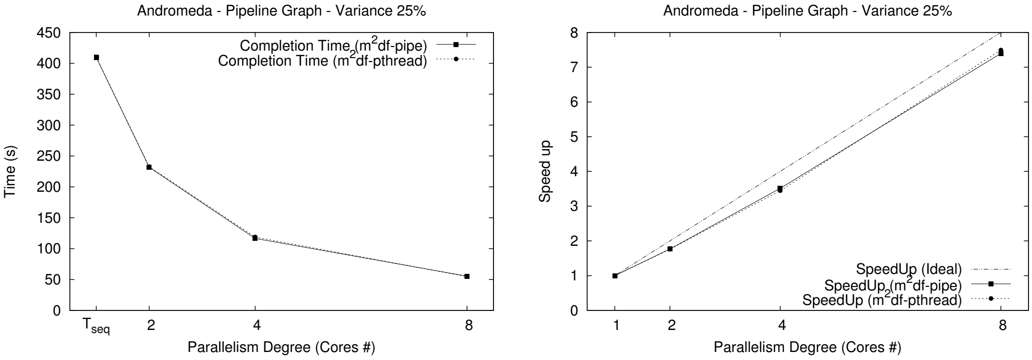
(d) Speed up[1]

Figure 6.5: Results of Generic Graph computation, with variance of 25%.



(a) Completion time

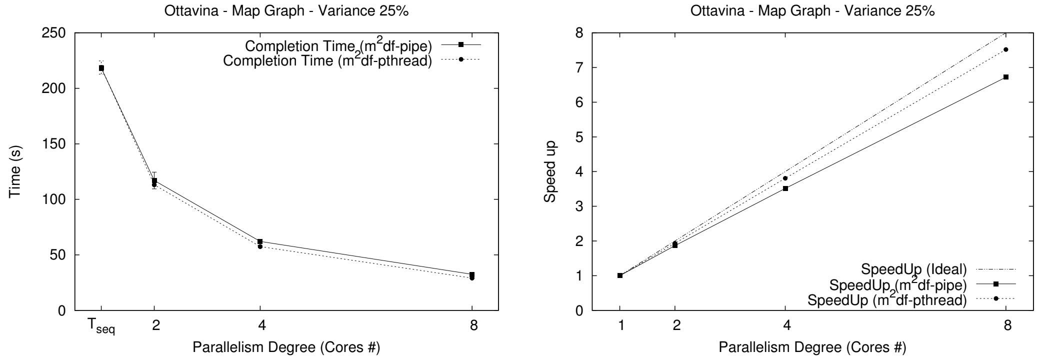
(b) Speed up[1]



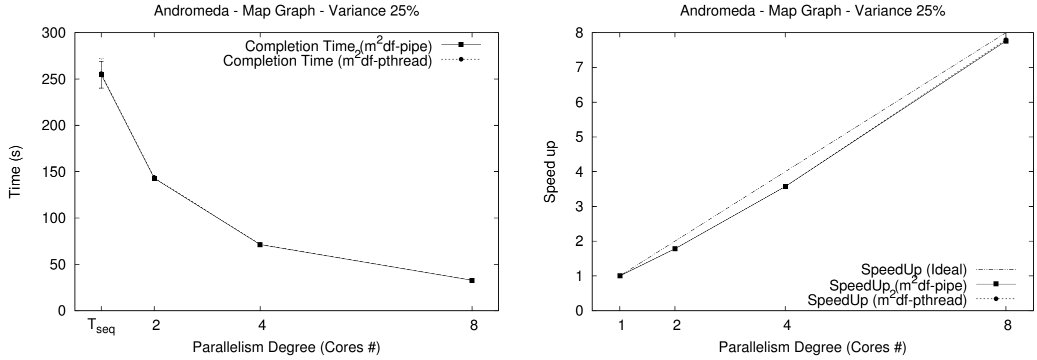
(c) Completion time

(d) Speed up[1]

Figure 6.6: Results of Pipeline Graph computation, with variance of 25%.



(a) Completion time (b) Speed up[1]



(c) Completion time (d) Speed up[1]

Figure 6.7: Results of Map Graph computation, with variance of 25%.

## Matrix power raising

In order to test the behaviour of the support with applications involving data transfers from the main memory and the exploitation of the memory hierarchy we implemented the power raising of a matrix.

This computation is structured as a pipeline, in which each stage performs a matrix product, as shown in Figure 6.8.

Each `mul` stage receives as input two matrices, `A` and `B`. It computes a naive matrix multiplication $B = B \times A$ and produces as output the matrix A and the updated matrix `B`.

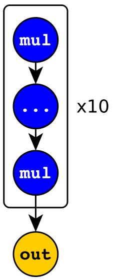


Figure 6.8: Graph structure of the matrix power raising.

In order to guarantee correctness at the beginning of the computation (*i.e.* the first stage's input) we have $B = A$.

We performed tests for both fine and coarse grain tasks. Fine grain tests consist in streams of 2048 $128 \times 128$ matrices while coarse grain computations consist in streams of 64 $640 \times 640$ matrices. All tests have been performed using optimized compilation.[2]

Also in this case we evaluated the single shot execution time, in order to have an idea of the computation grain.

| | `ottavinareale` | `andromeda` |
|---|---|---|
| $128 \times 128$ Matrix | $0.1491 \pm 0.0069$ | $0.0478 \pm 0.0023$ |
| $640 \times 640$ Matrix | $23.2389 \pm 0.8902$ | $27.0391 \pm 0.0759$ |

Table 6.2: Single shot execution times for matrix power raising, *in seconds.*

Figure 6.9 shows the fine grain test results and Figure 6.10 shows the coarse grain test results.

For fine grain computations both `ottavinareale` and `andromeda` doesn't suffer from cache misses. `ottavinareale` overscales and `andromeda` scales about 85/90% of the ideal speed up. Moving to coarse grain computations `andromeda` maintains the same performances of the fine grain case while `ottavinareale` scales about 60% of the ideal speed up for a degree of 8.

We can impute this difference in the two machines performance to the different cache sizes and policies implemented. Indeed, as described in Section 6.1, `andromeda` has a bigger cache than `ottavinareale`, moreover `andromeda` adopts the QuickPath Interconnect technology, which makes communications between the caches of the two processors faster.

In order to confirm this fact we performed further tests in which we studied the efficiency trend as a function of the matrix size, and hence of the cache exploitation. Figure 6.11 shows the efficiency trend for a fixed parallelism degree of *8*, on `ottavinareale`.

### Cholesky Decomposition

The Cholesky decomposition is the decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose:

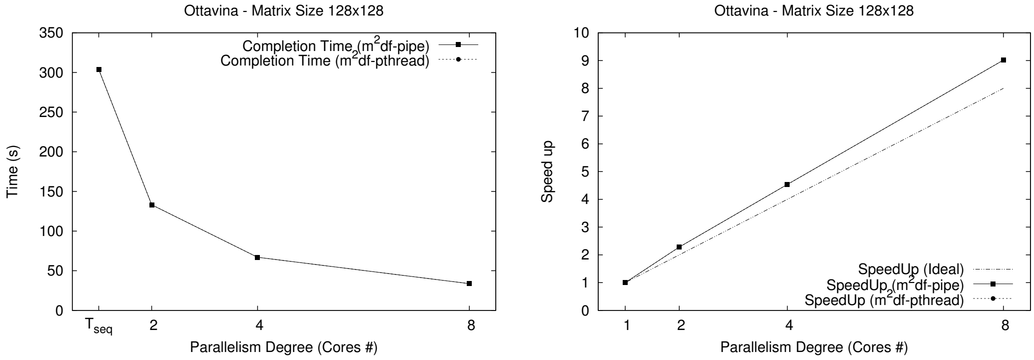
$$A = LL^T.$$

The Cholesky decomposition is mainly used for the numerical solution of linear equations $Ax = b$. Real-world applications often generate systems having the $A$ matrix positive-definite.

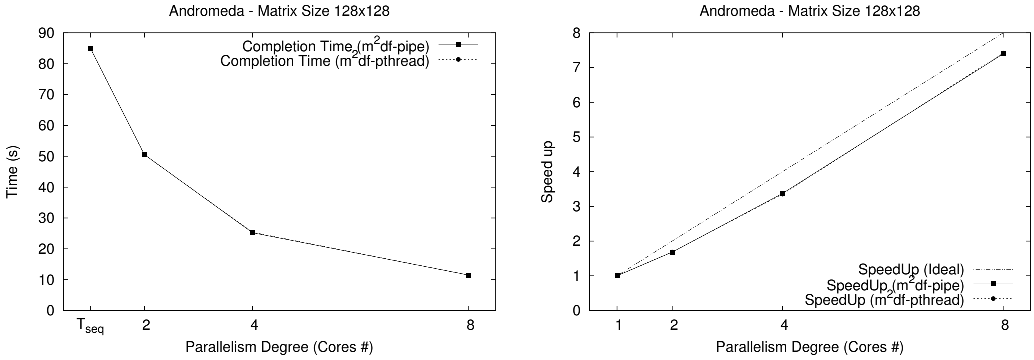
As a test algorithm we took the tiled implementation exposed in [15]. This version behaves exactly as the blocked Cholesky decomposition algorithm but, in this case the matrix is processed by tiles.

---

[2] `g++ -O3` compiler option

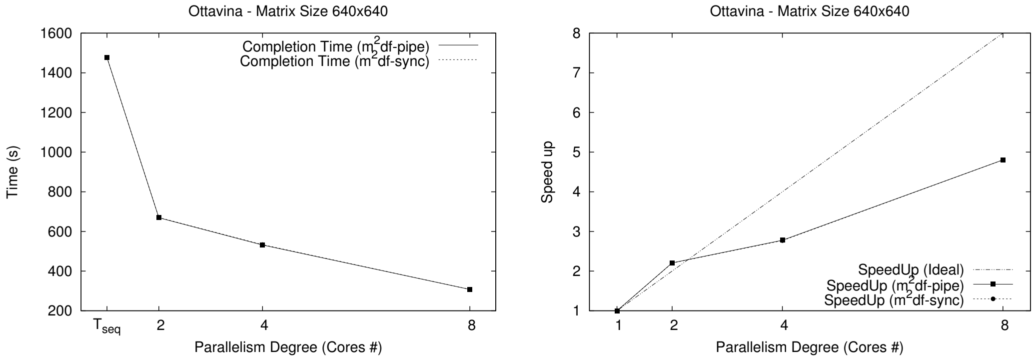


(a) Completion time (b) Speed up[1]

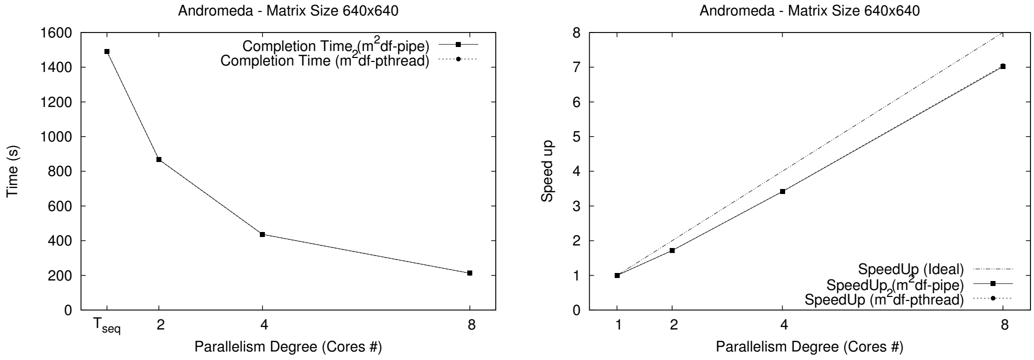


(c) Completion time (d) Speed up[1]

Figure 6.9: Test results for the power raising of a 128 × 128 matrices stream.



(a) Completion time (b) Speed up[1]



(c) Completion time (d) Speed up[1]

Figure 6.10: Test results for the power raising of a 640 × 640 matrices stream.

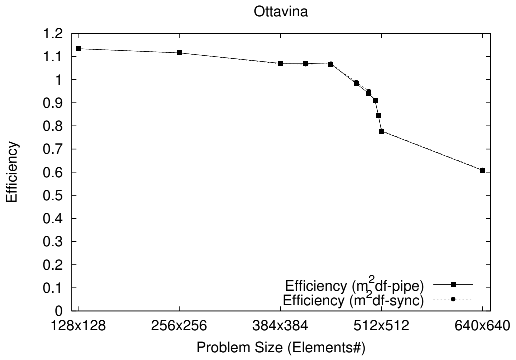


Figure 6.11: Trend of the efficiency[1] with respect to the problem size for a fixed parallelism degree of *8*.

```
    for(n = 0; n < k-1; n++)
        A[k][k] ← DSYRK(A[k][n], A[k][k])
    A[k][k] ← DPOTRF(A[k][k])

    for(m = k+1; m < TILES; m++) {
        for(n = 0; n < k-1; n++)
            A[m][k] ← DGEMM(A[k][n], A[m][n], A[m][k])
        A[m][k] ← DTRSM(A[k][k], A[m][k])
    }
}
```

Listing 6.1: Pseudocode of the tiled Cholesky decomposition.

A tile is a piece of the input matrix, which is stored contiguously in memory. This representation format is referred ad Block Data Layout [14]. Listing 6.1 shows the pseudocode of the algorithm utilized.

The algorithm relies on some BLAS and Lapack routines. Specially:

**DPOTRF** is a Lapack routine which performs the Cholesky factorization of a diagonal tile, and overrides the lower triangular part of the input tile with the result elements;

**DSYRK** is a BLAS routine which applies a symmetric rank-k update. In Listing 6.1 it is used to update a diagonal tile, departing from the tiles to the left of it;

**DGEMM** is a BLAS routine which performs a general matrix-matrix product. In Listing 6.1 it is used to update an off-diagonal tile, departing from the tiles to the left of it;

**DTRSM** is a BLAS routine which performs a triangular solve. In Listing 6.1 it is used to update an off-diagonal tile, departing from the tile above of it.

The average single shot execution time, *i.e.* the completion time of the single element of the stream, was measured as $0.9429 \pm 0.0092$ *seconds*.

Figure 6.12 shows the graph corresponding to the execution of this algorithm on a matrix composed of 5x5 tiles. We can observe that even though the number of tiles is low the graph structure is quite far from being intuitive.

Figure 6.13 shows the experimental results of the Cholesky decomposition on `andromeda`. All tests have been performed using optimized compilation.[2]

We can observe how the factorization follows the ideal speed up trend.

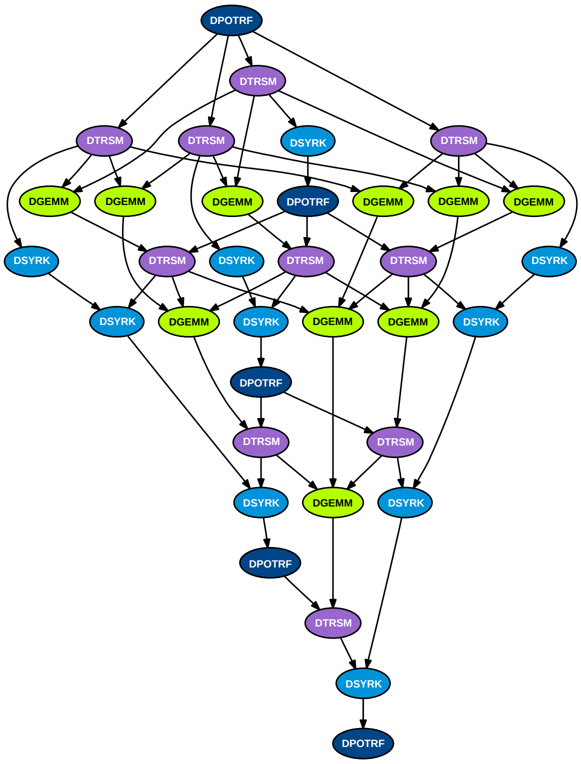


Figure 6.12: Task graph of the Cholesky decomposition on a 5x5 tiles matrix. *Figure taken from [15].*

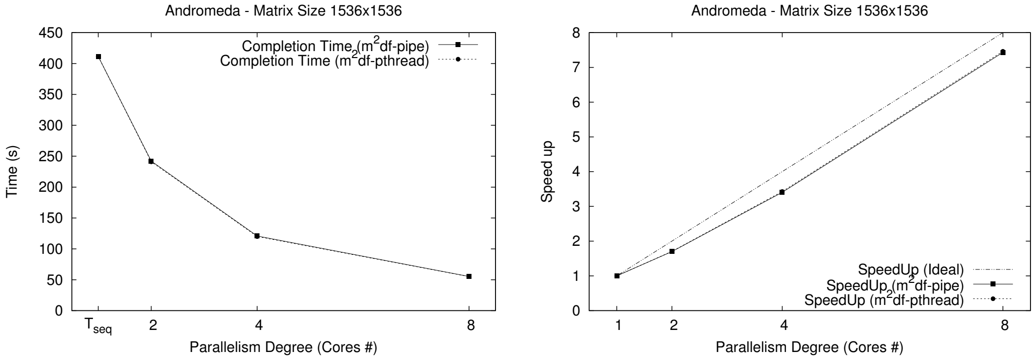
### QR Factorization

The QR factorization is a decomposition of a matrix A into the product of an orthogonal matrix by an upper triangular matrix:

$$A = QR,$$

where $Q$ is the orthogonal matrix and $R$ is the upper triangular matrix.

It is demonstrated that this factorization is applicable to all invertible matrices. This decomposition is generally used in the solving systems of



(a) Completion time

(b) Speed up[1]

Figure 6.13: Results of Cholesky Decomposition on `andromeda`.

linear equations, and is the basis for the *QR eigenvalues algorithm*.

Also in this case we used the tiled version of the algorithm. It behaves as the blocked version, except that the matrix is processed by tiles. Listing 6.2 shows the pseudocode of the algorithm.

```
for(k = 0; k < TILES; k++) {
    A[k][k], T[k][k] ← DGEQRT(A[k][k])
    for(m = k+1; m < TILES; m++)
        A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m
            ][k])
    for(n = k+1; n < TILES; n++) {
        A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
        for(m = k+1; m < TILES; m++)
            A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n],
                A[m][n])
    }
}
```

Listing 6.2: Pseudocode of the tiled QR factorization.

This algorithm is based on some routines relying on BLAS and Lapack functionalities:

**DGEQRT** performs the QR factorization of a diagonal tile, storing the $R$ factor on the upper triangular part of the input tile, and the Householder reflectors $V$ on the lower triangular part of the input tile. This routine also produces an auxiliary upper triangular matrix $T$ containing a compact representation of the reflectors;

**DTSQRT** updates two tiles of the input matrix by applying the QR factorization on a matrix obtained merging the $R$ factor calculated by DGEQRT or a previous call to DTSQRT and a tile below it. This routine overrides the $R$ factor, the sub-diagonal tile with the Householder reflectors $V$ and produces an auxiliary tile $T$ containing the compact representation;

**DLARFB** applies the reflectors calculated by DGEQRT $V$, along with the matrix $T$ to a tile to the right of it;

**DSSRFB** applies the reflectors calculated by DTSQRT $V$, along with the matrix $T$ to two tiles on the right of the factorized one.

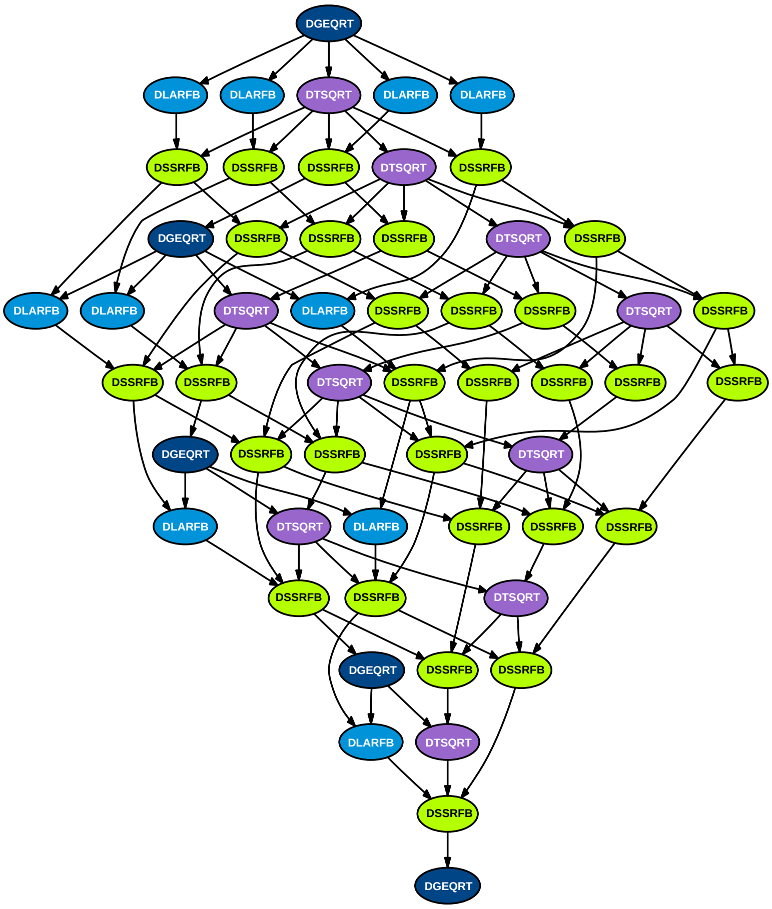
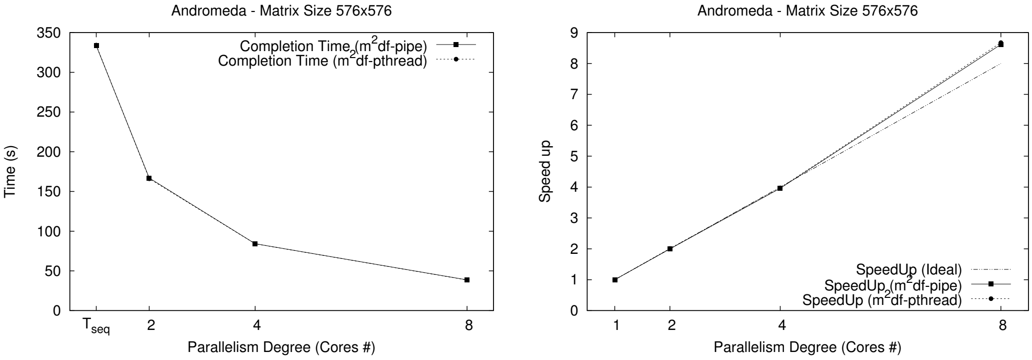


Figure 6.14: Task graph of the QR factorization on a 5x5 tiles matrix. *Figure taken from [15].*

Also in this case, in order to have an idea of the computation grain, we measured the average completion time of the single instance of the graph as $0.6471 \pm 0.0041$ *seconds*.

Figure 6.14 shows the graph resulting from the application of this algorithm to a 5x5 tiles matrix. In this case the resulting graph is much more complex than the Cholesky one.

Figure 6.15 shows the experimental results on `andromeda`. Tests ran using optimized compilation.[2]



(a) Completion time

(b) Speed up[1]

Figure 6.15: Results of the QR Factorization on `andromeda`.

The Figure shows how the speed up follows the linear trend, overscaling for higher parallelism degrees.

## 6.3 Comparison with OpenMP

In order to deepen the study of the `m`$^2$`df`'s behaviour we tested a very simple algorithm, comparing the results with a standard industrial product such as OpenMP.

Unlike `m`$^2$`df` OpenMP adopts a fork/join model. In such a model a new thread pool is forked when a parallel section arises. The threads composing the thread pool are then joint together when the computation has done, as shown in Figure 6.16.

In order to compare the two models we choose a simple naive matrix product, in which the available parallelism is exploited at the outermost loop level. In this parallelization each row of the result matrix is computed in parallel.

Obtaining the same graph for more complex algorithms is not trivial and anyway we cannot be sure that the computation is parallelized the same way. Using such a simple algorithm we are sure that the computations have the same structure, hence comparing the results actually makes sense.

This parallelization results on a `#pragma omp for` directive on the outermost loop, in the OpenMP version, as shown in Listing 6.3, and in a map-type graph in the `m`$^2$`df` version, as shown in Figure 6.17.

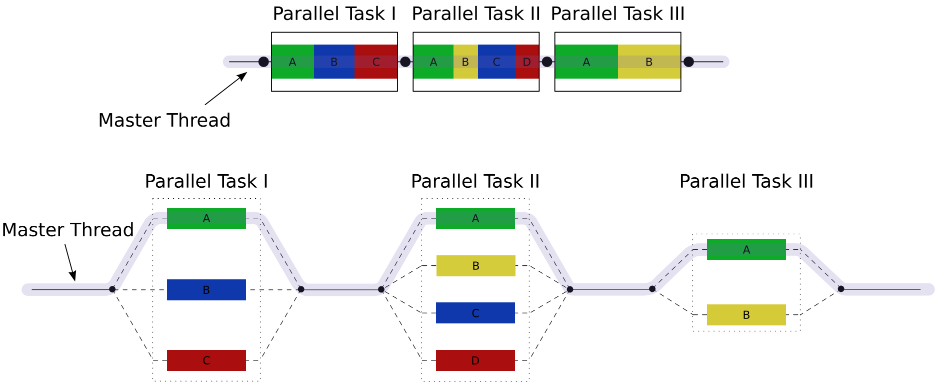


Figure 6.16: OpenMP multi-threading model of execution. *Figure taken from [24]*

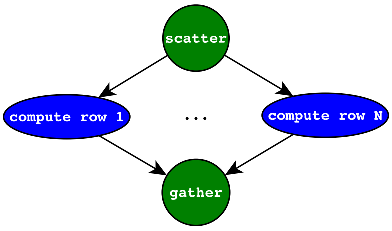


Figure 6.17: Graph structure for the matrix product parallelization.

For the OpenMP parallelization of this algorithm we used `static` scheduling since we are multiplying dense matrices. This fact makes the computation of different rows a naturally balanced computation. Furthermore the actual `m`$^{2}$`df` scheduling policy is a static one.

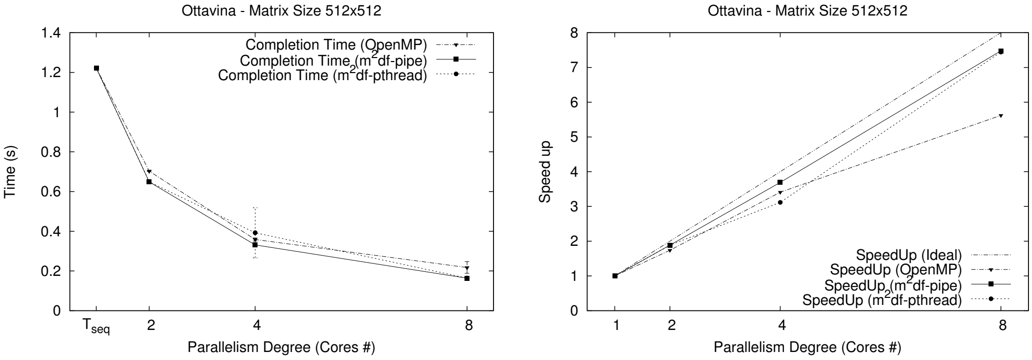
```
#pragma omp parallel \
default(none) num_threads(__np) \
shared(a, b, c) private (i, j, k)
{
#pragma omp for schedule(static) nowait
for(i = 0; i < MAT_SIZE; i++) {
    for(j = 0; j < MAT_SIZE; j++) {
        for(k = 0; k < MAT_SIZE; k++) {
            c[i*MAT_SIZE+j]+=(a[i*MAT_SIZE+k]*b[k*MAT_SIZE+j]);
        }
    }
}
}
```

Listing 6.3: OpenMP parallelization of matrix multiply

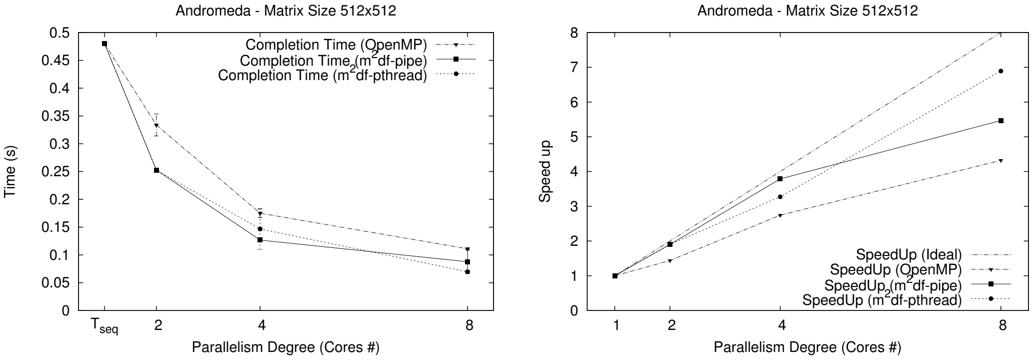
The comparison have been performed for different grains. As a fine grain computation we considered the product of two 512 × 512 elements matrices, as a medium grain computation we considered the product of two 1024×1024 elements matrices and as a coarse grain computation we considered the product of two 2048 × 2048 elements matrices.

Figures 6.18, 6.19 and 6.20 show the completion times and speed-ups comparing OpenMP with the two versions of `m²df`.

Figure 6.21 shows the efficiency trend with respect to the problem size for a fixed parallelism degree of *4*. Also for these tests we used optimized compilation.[2]

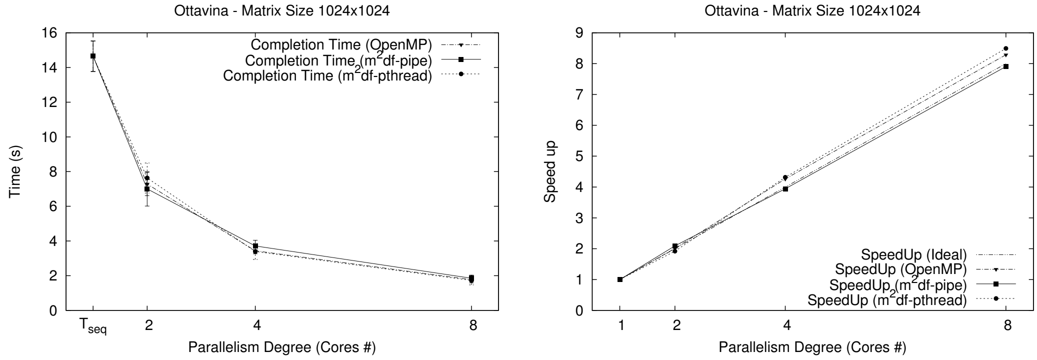


(a) Completion time (b) Speed up[1]



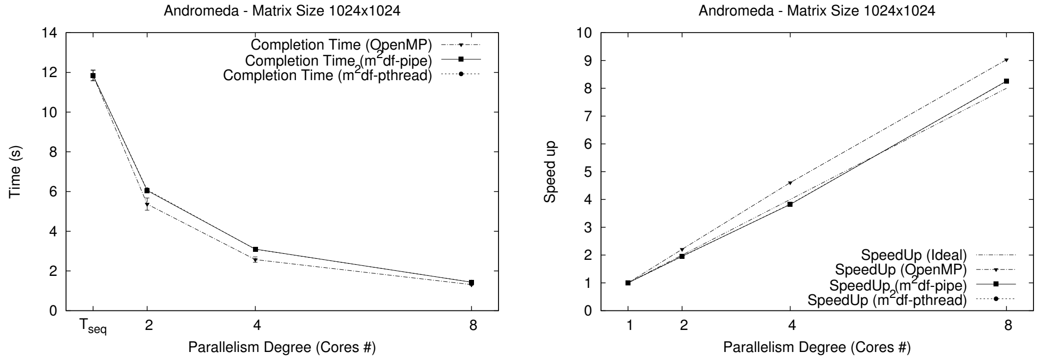
(c) Completion time (d) Speed up[1]

Figure 6.18: Test results for fine grained matrix product.



(a) Completion time

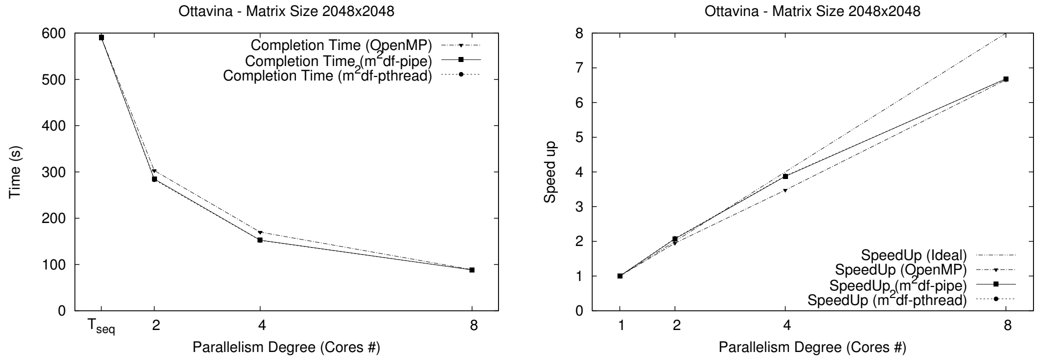
(b) Speed up[1]



(c) Completion time

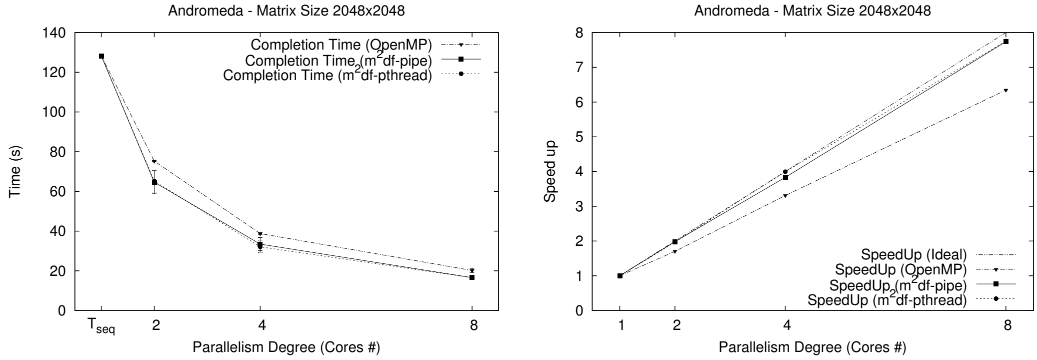
(d) Speed up[1]

Figure 6.19: Test results for medium grained matrix product.



(a) Completion time

(b) Speed up[1]



(c) Completion time

(d) Speed up[1]

Figure 6.20: Test results for coarse grained matrix product.

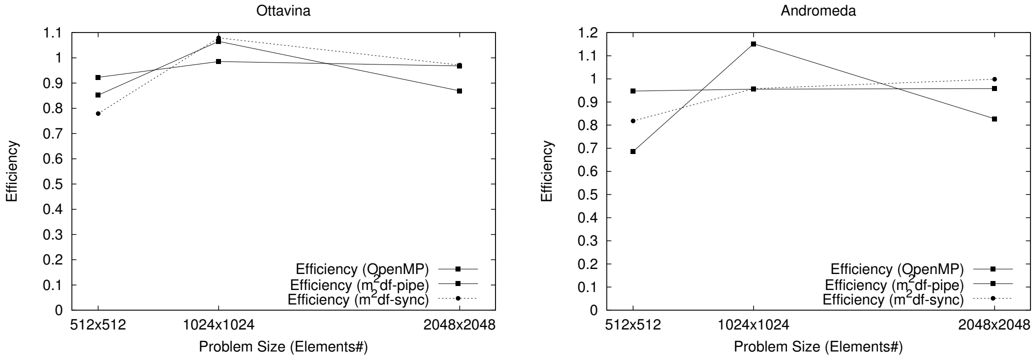


Figure 6.21: Trend of the efficiency[1] with respect to the problem size for a fixed parallelism degree of *4*.

In these Figures we can appreciate how `m`$^2$`df` behaves, compared to OpenMP. For both fine and coarse grain computations `m`$^2$`df`overcomes OpenMP, while for medium grain computations it doesn't happen. In spite of this the speed up always follows the linear trend.

## 6.4 Experiment results

To recap, we presented several tests performed on `m`$^2$`df`:

- first we tested some synthetic application in order to verify the correct completion of the assigned computations and to study the framework behaviour with CPU-intensive computations not involving the memory;
- subsequently we tested an application involving memory accesses. This application consisted in a matrix power raising in which all stages computed a *cache-unaware* matrix multiply. We this application we studied the behaviour of `m`$^2$`df` for applications suffering of high cache-miss rates;
- then we implemented some real world applications, consisting in Cholesky and QR factorizations of a matrix. These applications used highly optimized routines, and a *cache-aware* implementation;
- finally we compared the `m`$^2$`df`'s performance with a standard state-of-the-art tool such as OpenMP.

All test have been performed on state-of-the-art multi-core processors and using standard tools.[3]

---

[3] Described in Section 6.1.

Test results shown that the proposed framework is able to reach significant speed up and has comparable with optimized industrial products. Nevertheless there are many possible optimizations and improvements to be studied.

# Chapter 7

# Conclusions and future work

In this thesis we presented $\mathtt{m}^2\mathtt{df}$, a prototype system implementing a multi-threaded Macro Data Flow interpreter. In this Chapter we recap the work that has been done and we suggest some possible extensions.

## 7.1 Contribution of the work

In this work we designed and implemented a multi-threaded system, which implements a Macro Data Flow interpreter targeting multi-cores.

### 7.1.1 The idea

Looking to some state-of-the-art systems we targeted Macro Data Flow as a viable alternative for efficiently exploiting off-the-shelf multi-core processors.

We designed the overall structure of a multi-threaded system composed of entities which interact together in order through the shared memory communication mechanism in order to finalize the computation.

The main actors of the system are a set of *interpreter threads* which execute the tasks assigned to them and a *scheduler thread*, which is in charge of distributing the tasks composing an instance among the interpreters.

Along with these entities have been designed a set of supporting structures for making the system interact with the user.

The resulting system allows the user to create different graphs representing computations and to submit several instances of the graphs for the executions in a streamed fashion.

The system is capable to automatically determine the number of interpreters to launch or, alternatively, to get it from the user.

The user is only aware of the outermost structure of the graph (since he creates it) but, once an instance is submitted for the execution it is entirely handled by the scheduler.

### 7.1.2 The implementation

After the design phase we discussed implementative choices relating the language and the supporting tools.

In order to have high portability we choose C++ as implementation language and the POSIX standard as portability layer.

We implemented a run-time support abstracting from the communication mechanism, relegating the communication concerns to a class `shared_queue`.

We gave two different implementations of this class. The first implementation relies on the `pthread` synchronization primitives and the second one relies on the `pipe` mechanism.

In order to have a correct working of the communication mechanism we implemented other supporting structures such as a semaphore.

During the implementation we faced problems related to the handling of the threads. In order to have the interpreters working independently it was necessary to enforce each interpreter to run on the same core for all of its lifetime.

On the contrary, the scheduler thread is allowed to run on any free core. In this way the scheduler takes over the idle interpreters and distributes the ready tasks, according to the scheduling policy.

### 7.1.3 The results

We tested $\mathtt{m}^2\mathtt{df}$ with different applications on state-of-the-art Intel multicores.

We used several kind of applications spacing from synthetic applications to complex matrix factorization algorithms.

We studied the run-time support behaviour with respect to the computation grain and to the memory hierarchy exploitation.

First, we tested $\mathtt{m}^2\mathtt{df}$ with synthetic CPU-dominant computations.

Second, we tested the system with the matrix power raising, a synthetic memory-dominant application.

Then we implemented some real-world applications such as the Cholesky and QR decomposition of a matrix.

Finally we compared $\mathtt{m}^2\mathtt{df}$ with a standard tool such as OpenMP using a simple naive matrix multiply algorithm with different computation grains, in order to be sure that the parallelization was equal in the two cases.

Experimental results shown how the proposed framework has good performances, comparable with standard state-of-the-art tools.

## 7.2 Future work

There are many ways in which the proposed system can be improved. Furthermore this thesis leaves some open research trend.

First of all a complete system should be implemented. `m`$^2$`df` was conceived to be the back-end of a compiler which builds graphs out of a structured application.

In order to remark this point Listing 7.1 shows the creation of the tiled QR graph used to test `m`$^2$`df`. Looking to the sequential version of the same algorithm, shown by Listing 6.2, we can notice structural similarities between the two codes.

A compiler could be able to derive such a graph following appropriate rules to find the data dependencies during the pre-processing phase.

Another possible development of the system could be a skeleton library built upon the system. Other research groups already demonstrated how skeletons can be reduced to MDF graphs [17, 18].

Other open trends relate to the scheduling of MDF graphs.

`m`$^2$`df` indeed implements the scheduling policy in a single point of the code. This design choice makes it simple to change policy.

Different scheduling policies should be implemented and used to test the performance with respect to this factor.

Scheduling policies could be developed targeting some non-functional aspects of the computation such as:

- load balancing with respect to "big" tasks;
- fault tolerance;
- smart cache hierarchy exploitation [13, 20].

The support should be tested into a many-cores architecture. In these environments a single scheduler could represent a performance bottleneck.

To overcome this (potential) problem hierarchic scheduling should be a solution. In order to implement such a hierarchy it is sufficient to modify the scheduler loop behaviour.

From the usability viewpoint functionalities related to the graph handling could be added, such as the merging of graph pre-existing graphs.

This functionality allow the user to create pieces of basic graphs and instantiate them into other graphs. An example could be represented by the matrix power raising: the user should be able to create a parallel matrix multiply graph and then connect different instances of the graph in a pipeline fashion.

```
    unsigned dgeqrt_id = g.addNode(test::dgeqrt);
    if(k > 0) {g.addEdge(dssrfb_ids[k][k], dgeqrt_id);}

    prec_id = dgeqrt_id;
    vector < unsigned > dtsqrt_ids;
    for(int m = k+1; m < n_tiles; m++) {
        unsigned dtsqrt_id = g.addNode(test::dtsqrt);
        g.addEdge(prec_id , dtsqrt_id);
        if(k > 0) {g.addEdge(dssrfb_ids[m][k], dtsqrt_id);}
        prec_id = dtsqrt_id;
        dtsqrt_ids.push_back(dtsqrt_id);
    }

    for(int n = k+1; n < n_tiles; n++) {
        unsigned dlarfb_id = g.addNode(test::dlarfb);
        g.addEdge(dgeqrt_id , dlarfb_id);
        if(k > 0){ g.addEdge(dssrfb_ids[k][n], dlarfb_id);}

        prec_id = dlarfb_id;
        unsigned i = 0;
        for(int m = k+1; m < n_tiles; m++) {
            unsigned dssrfb_id = g.addNode(test::dssrfb);
            g.addEdge(dtsqrt_ids[i], dssrfb_id);
            g.addEdge(prec_id , dssrfb_id);
            if(k > 0) g.addEdge(dssrfb_ids[m][n], dssrfb_id);

            dssrfb_ids[k][n] = dssrfb_ids[m][n] = dssrfb_id;
            prec_id = dssrfb_id;
            i++;
        }
    }
}
```

Listing 7.1: Creation of the tiled QR graph.

## 7.3 Conclusions/Summary

To summarize this thesis:

- addresses Macro Data Flow computing model as a viable alternative for exploiting state-of-the-art multi-cores;
- discusses the design phase of such an interpreter;
- deepens implementation aspects of the system;

- evaluate its performances on state-of-the-art multi-cores comparing it with serial versions and similar tools.

# Appendix A

# Source code

## m$^2$df

Next Listings show the source code for the user interface routines.

m2df.h

```
#ifndef M2DF_H
#define M2DF_H 1

#include "m2df_graph.h"
#include "m2df_interpreter.h"
#include "m2df_queues.h"
#include "m2df_scheduler.h"
#include "m2df_semaphore.h"

#include <climits>
#include <unistd.h>

namespace m2df
{
    /**
     * \brief This function tunes the rts. Gets the cores number
          and creates the scheduler and the interpreters.
     * \param __interpreters_n [Default value 0(meaning cores
         number)] Number of interpreters to run with.
     */
    void tune(unsigned __interpreters_n = 0) throw(
        MultipleTuneException);
    /**
     * \brief This function launches the parallel execution of
         the graphs submitted to the rts.
     */
    void start(unsigned __interpreters_n = 0);
    /**
     * \brief This function executes the termination phase, it
         __must__ be called before terminating the execution.
     */
    void finalize();
```

```
    /**
     * \brief This function returns the number of cores available in the machine.
     * \return The number of cores which are being used (if RTS has been tuned).
     */
    unsigned get_cores_number();
    /**
     * \brief This function returns the effective parallelism degree exploited by the RTS.
     * \return The number of cores which are being used (if RTS has been tuned).
     */
    unsigned get_parallelism_degree();
}

#endif //M2DF_H
```

m2df.cpp

```
#include "m2df.h"

namespace m2df
{
namespace global
{
unsigned processors_number, interpreters_number;
Scheduler *scheduler_addr = NULL;
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}

void tune(unsigned __interpreters_n) throw(MultipleTuneException)
{
    if(global::scheduler_addr != NULL) throw MultipleTuneException();
    #ifdef M2DF_DEBUG
        pthread_mutex_init(&global::comm_mutex, NULL);
    #endif

    global::processors_number = sysconf(_SC_NPROCESSORS_ONLN);
    if(global::processors_number == -1) global::processors_number = 1; //if there is some error, goes sequential.
    global::interpreters_number = global::processors_number;

    if(__interpreters_n == 0)
        global::scheduler_addr = new Scheduler(global::processors_number);
    else {
        global::scheduler_addr = new Scheduler(__interpreters_n);
```

```
        global::interpreters_number = __interpreters_n;
    }
}

void start(unsigned __interpreters_n) //actually, it is a non-blocking call
{
    if(global::scheduler_addr != NULL) global::scheduler_addr->run();
    else tune(__interpreters_n);
}

void finalize()
{
    if(global::scheduler_addr != NULL) {
        global::scheduler_addr->enableCompletion();

        void *dummy;
        pthread_join(global::scheduler_addr->GetThreadInfo(), &dummy);
        delete global::scheduler_addr;
        global::scheduler_addr = NULL;
    }
    #ifdef M2DF_DEBUG
    pthread_mutex_destroy(&global::comm_mutex);
    #endif
}

unsigned get_cores_number()
{
    if(global::scheduler_addr != NULL) return (global::processors_number);
    else return 0;
}

unsigned get_parallelism_degree()
{
    unsigned par_deg;
    if(global::interpreters_number < global::processors_number)
        par_deg = global::interpreters_number;
    else
        par_deg = global::processors_number;

    return par_deg;
}

}
```

# Abstract Node

Next Listings show the implementation of the abstract node structure.

`m2df_abstract_node.h`

```
#ifndef ABSTRACT_NODE_H
#define ABSTRACT_NODE_H 1
#include "m2df_debug.h"
#include "m2df_task.h"

#include <pthread.h>
#include <vector>

using namespace std;

namespace m2df
{

/**
 * \struct _abstract_node
 * \author Lorenzo Anardu
 * \date 09/12/2010
 * \file m2df_abstract_node.h
 * \brief Abstract representation of a MDF task.
 */
struct _abstract_node {
private:
    unsigned *copies_cnt; //counts the copies of an instance
    unsigned uid;  //unique_id
    pthread_mutex_t _mutex;

    /**
     * \brief Creates a new unique id for the node.
     * \return Unique id value.
     */
    unsigned getNewUid(); //gets a new uid
public:
    unsigned tid, ready_cnt; //task_id, ready counter
    vector<unsigned> consumers; //consumer nodes of the node
    task *addr; //concrete represantion

    /**
     * \brief Default constructor, does nothing.
     */
    _abstract_node();
    /**
     * \brief Constructor which initializes a new abstract_node
         with task_id.
     * \param __tid Id of the task in the graph.
     */
    _abstract_node(unsigned __tid);
    /**
     * \brief Copy constructor.
```

```
     * \param __n Abstract node to copy.
     */
    _abstract_node(const _abstract_node& __n);
    /**
     * \brief Destructor.
     */
    virtual ~_abstract_node();

    /**
     * \brief Assignment operator.
     * \param __n Abstract node to copy.
     */
    _abstract_node& operator=(const _abstract_node& __n);

    /**
     * \brief Getter: gets the unique id of the node.
     * \return Unique id value.
     */
    unsigned getUid() const {
        return uid;
    }
};

typedef struct _abstract_node abstract_node;


inline unsigned _abstract_node::getNewUid()
{
    //FIXME: add a mutex lock on the static variable
    static unsigned new_uid = 0;
    return new_uid++;
}

inline _abstract_node::_abstract_node()
    : uid(0), tid(0), ready_cnt(0), addr(NULL), consumers()
{ }

inline _abstract_node::_abstract_node(unsigned __tid)
    : tid(__tid), ready_cnt(0), addr(NULL)
{
    pthread_mutex_init(&_mutex, NULL);
    copies_cnt = new unsigned(1);
    uid = getNewUid();
}

} //end m2df namespace

#endif //ABSTRACT_NODE_H
```

`m2df_abstract_node.cpp`

```
#include "m2df_abstract_node.h"

namespace m2df
{
namespace global
{
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}


_abstract_node::_abstract_node(const _abstract_node& __n)
    : tid(__n.tid), ready_cnt(__n.ready_cnt), consumers(__n.consumers), addr(__n.addr)
{
    copies_cnt = __n.copies_cnt;
    _mutex = __n._mutex;
    pthread_mutex_lock(&_mutex);
    (*copies_cnt)++;
    pthread_mutex_unlock(&_mutex);
    uid = __n.getUid();
}

_abstract_node::~_abstract_node()
{
    consumers.clear();
    pthread_mutex_lock(&_mutex);
    --(*copies_cnt);
    unsigned copies_n = *copies_cnt;
    pthread_mutex_unlock(&_mutex);

    if(copies_n == 0) {
        delete copies_cnt;
        if(addr != NULL)
            delete addr;
        pthread_mutex_destroy(&_mutex);
    }
}

_abstract_node& _abstract_node::operator=(const _abstract_node& __n)
{
    if(this == &__n) return *this;

    uid = __n.getUid();
    tid = __n.tid;
    ready_cnt = __n.ready_cnt;
    consumers = __n.consumers;
    copies_cnt = __n.copies_cnt;
    _mutex = __n._mutex;
```

```
    pthread_mutex_lock(&_mutex);
    (*copies_cnt)++;
    pthread_mutex_unlock(&_mutex);

    addr = __n.addr;

    return *this;
}
} //end m2df namespace
```

## Completion

Next Listing shows the implementation of the completion structure.

`m2df_completion.h`

```
#ifndef COMPLETION_H
#define COMPLETION_H 1
#include "m2df_debug.h"

namespace m2df
{

/**
 * \struct _completion
 * \author Lorenzo Anardu
 * \date 09/12/2010
 * \file m2df_completion.h
 * \brief Completion of a task.
 */
struct _completion {
    unsigned gid, tid;

    /**
     * \brief Default constructor, does nothing.
     */
    _completion() : gid(0), tid(0) {}
    /**
     * \brief Constructor which initializes a new abstract_node
         with task_id.
     * \param __task_id Id of the task.
     */
    _completion(unsigned __gid, unsigned __tid) : gid(__gid),
        tid(__tid) {}
};

typedef struct _completion completion;

} //end m2df namespace

#endif //COMPLETION_H
```

## Debug

Next Listings show the configuration flags declaration, and the declaration of the mutex used for safe debug communications.

m2df_debug.h

```
#ifndef DEBUG_H
#define DEBUG_H 1
#include <iostream>
#include <pthread.h>

/*************|--->DEFINITIONS<---|*************
 * M2DF_DEBUG            Makes debug messages to be printed.
 * M2DF_TRACING          Makes the interpreters to keep trace of the computed tasks.
 * M2DF_PIPES_VERSION    Makes m2df to be compiled with communication system based on the pipe mechanism.
 * M2DF_SYNC_VERSION     Makes m2df to be compiled with communication system based on the pthread synchronization primitives.
 *
 * Note: M2DF_PIPES_VERSION and M2DF_SYNC_VERSION are mutually exclusive.
 *****************|--->END<---|*****************/

//#define M2DF_DEBUG
//#define M2DF_TRACING
#define M2DF_PIPES_VERSION

#ifndef M2DF_PIPES_VERSION
#define M2DF_SYNC_VERSION
#endif

#endif //DEBUG_H
```

m2df_debug.cpp

```
/**
    m2df is a run time support allowing the efficient exploitation of
    multi-core processors, through multi-threading and macro data flow
    techinque. It relies on POSIX functionalities.

    Copyright (C) 2011  Lorenzo Anardu (lorenzo.anardu AT live DOT com)

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
```

```
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
**/
#include "m2df_debug.h"

namespace m2df
{
namespace global
{
#ifdef M2DF_DEBUG
    pthread_mutex_t comm_mutex; //mutex used for safe debug communications
#endif
} //end global namespace

} //end m2df namespace
```

## Exceptions

Next Listing shows the m$^{2}$df exceptions implementation.

`m2df_exceptions.h`

```
#ifndef EXCEPTION_H
#define EXCEPTION_H
#include <cerrno>
#include <string>

using namespace std;

namespace m2df
{

class Exception {
protected:
    string cause;

//  Exception() { }
public:
    virtual const char *what() {return "";}
```

```
};

/*
 * GRAPH EXCEPTIONS
 */
class BadEdgeException : public Exception
{
public:
    const char *what() {
        return "Inserted␣edge␣is␣not␣correct.";
    }
};

class NotNormalizedException : public Exception
{
public:
    const char *what() {
        return "Graph␣is␣not␣normalized␣(single␣entry␣and␣single␣exit␣point).";
    }
};

/*
 * QUEUES EXCEPTIONS
 */
class EmptyQueueException : public Exception
{
public:
    const char *what() {
        return "You␣are␣trying␣to␣pop␣something␣from␣an␣empty␣queue.";
    }
};

class FullQueueException : public Exception
{
public:
    const char *what() {
        return "You␣are␣trying␣to␣push␣something␣in␣a␣full␣queue.";
    }
};

/*
 * THREAD EXCEPTIONS
 */
class ThreadForkException : public Exception
{
//  string mesg;
public:
    ThreadForkException() {
        cause = "Error␣in␣pthread␣create.";
    }
    ThreadForkException(int __error) {
```

```
        switch(__error) {
            case EAGAIN: cause = "EAGAIN";
            break;
            case EINVAL: cause = "EINVAL";
            break;
            case EPERM: cause = "EPERM";
        }
        cause += "␣error␣in␣pthread_create.";
    }

    const char *what() const{
        return cause.c_str();
    }
};

class ThreadSchedulingException : public Exception
{
    string mesg;
public:
    ThreadSchedulingException(int __error) {
        switch(__error) {
            case EFAULT: mesg = "EFAULT";
            break;
            case EINVAL: mesg = "EINVAL";
            break;
            case ESRCH: mesg = "ESRCH";
        }
        mesg += "␣error␣in␣pthread_setaffinity_np.";
    }

    const char *what() {
        return mesg.c_str();
    }
};

/*
 * MISCELLANEOUS EXCEPTIONS
 */
class BadAllocException : public Exception
{
public:
    const char *what() {
        return "Error␣in␣allocating␣memory␣(no␣space␣left).";
    }
};

class MultipleTuneException : public Exception
{
public:
    const char *what() {
        return "tune()␣called␣multiple␣times.";
    }
};

```

```
class PipeException : public Exception
{
public:
    const char *what() {
        return "Error in creating Pipe.";
    }
};

class PipeReadException : public Exception
{
public:
    const char *what() {
        return "Error in reading from Pipe.";
    }
};

class PipeWriteException : public Exception
{
public:
    const char *what() {
        return "Error in writing in Pipe.";
    }
};

} //end m2df namespace

#endif // EXCEPTION_H
```

## Graph

Next Listings show the graph class declaration and implementation.

m2df_graph.h

```
#ifndef GRAPH_H
#define GRAPH_H 1
#include "m2df.h"
#include "m2df_abstract_node.h"
#include "m2df_exceptions.h"
#include "m2df_scheduler.h"
#include "m2df_utils.h"

#include <cstring>
#include <queue>
#include <utility>
#include <vector>

using namespace std;

namespace m2df
{
```

```
/**
 * \class Graph
 * \author Lorenzo Anardu
 * \date 09/12/2010
 * \file m2df_graph.h
 * \brief This class represents a MDF graph.
 */
class Graph {
    bool checked;
    unsigned graph_id;
    unsigned node_id, inst_id, root_id;
    queue< Edge > edges;
    vector<abstract_node> abstr_graph;
    vector<void**(*)(void**)> routines;

    unsigned getId();
    /**
     * \brief Check if the graph has an unique input node and a
         unique output node.
     * \return true, if the graph can be instanciated; false
         otherwise.
     */
    bool check();
public:
    /**
     * \brief Default constructor: does noting.
     */
    Graph();
    /**
     * \brief Destructor.
     */
    virtual ~Graph();

    /**
     * \brief Adds a node to the graph, and associates it the
         code to be executed.
     * \param __routine Code to be executed.
     * \return The id of the node.
     */
    unsigned addNode(void**(*__routine)(void**));

    /**
     * \brief Adds an edge to the graph, linking two existing
         nodes.
     * \param __from Id of the origin node (returned from
         addNode).
     * \param __to Id of the destination node (returned from
         addNode).
     */
    void addEdge(unsigned __from, unsigned __to) throw(
        BadEdgeException);

    /**
     * \brief Creates a new instance of the graph, and submits
```

```
            it to the RTS.
        * \param __input_data [Default value = NULL]Data to be passed to the root of the gaph.
        */
       unsigned submitInstance(void **__input_data = NULL, void **__res_placeholder = NULL) throw(NotNormalizedException);
};

} //emn medf namespace

#endif //GRAPH_H
```

m2df_graph.cpp

```
#include "m2df_graph.h"

using namespace std;

namespace m2df
{
namespace global
{
extern Scheduler *scheduler_addr;
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}

Graph::Graph()
    : node_id(0), inst_id(0), root_id(0), checked(false)
{
    graph_id = getId();
}

/********IMPORTANT********
 * Actually, Graph class doesn't explicitly
 * allocate memory. Hence there is no need
 * of an explicit copy constructor and
 * operator=. IF it will allocate some memory
 * IT HAS to be implemented (look @ other
 * classes for some implementation).
 ************************/

Graph::~Graph()
{
    abstr_graph.clear();
    routines.clear();
    while(!edges.empty())
        edges.pop();
}

unsigned Graph::addNode(void**(*__routine)(void**))
{
```

```
    abstract_node node(node_id++); //creates a new node

    abstr_graph.push_back(node);
    routines.push_back(__routine);
    checked = false;

    return node.tid;
}

void Graph::addEdge(unsigned __from, unsigned __to) throw(
    BadEdgeException)
{
    if(__from >= node_id || __to >= node_id) throw
        BadEdgeException();
    Edge edge;
    edge.first = __from;
    edge.second = __to;

    edges.push(edge);
    abstr_graph[__from].consumers.push_back(__to);
    abstr_graph[__to].ready_cnt++;
    checked = false;
}

unsigned Graph::submitInstance(void **__input_data, void **
    __res_placeholder) throw(NotNormalizedException)
{
    unsigned ret = 0;
    if(check() == false) throw NotNormalizedException();
    abstract_node *graph = new abstract_node[abstr_graph.size()
        ];
    for(unsigned i = 0; i < abstr_graph.size(); i++)
        graph[i] = abstr_graph[i]; //new copy

    for(unsigned i = 0; i < abstr_graph.size(); i++) { //creates
         the concrete representation of the graph (instance
        dependant)
        unsigned uid = abstr_graph[i].getUid();
        if(root_id == i && __input_data != NULL) //root node
            graph[i].addr = new task(inst_id, i, uid, routines[i
                ], abstr_graph[i].ready_cnt+1, abstr_graph[i].
                consumers.size(), __input_data);
        else
            graph[i].addr = new task(inst_id, i, uid, routines[i
                ], abstr_graph[i].ready_cnt, abstr_graph[i].
                consumers.size(), NULL);
    }

    Instance inst(graph_id, inst_id, graph, abstr_graph.size(),
        edges, root_id);

    if(m2df::global::scheduler_addr == NULL) tune();
    ((Scheduler*)m2df::global::scheduler_addr)->addInstance(inst
        );
```

```
    delete [] graph;
    ret = inst_id++;
    return ret;
}

unsigned Graph::getId()
{
    static unsigned next_id = 0;
    return (next_id++);
}

bool Graph::check()
{
    bool input_node = false, output_node = false;
    if(checked == true) return true;

    for(unsigned i = 0; i < abstr_graph.size(); i++) {
        if(abstr_graph[i].ready_cnt == 0)
            if(!input_node) { input_node = true; root_id = i; }
            else { input_node = false; break; }
        if(abstr_graph[i].consumers.size() == 0)
            if(!output_node) output_node = true;
            else { output_node = false; break; }
    }

    checked = (input_node && output_node);
    return checked;
}

}
```

## Instance

Next Listings show the instance class declaration and implementation.

`m2df_instance.h`

```
#ifndef INSTANCE_H
#define INSTANCE_H
#include "m2df_abstract_node.h"
#include "m2df_task.h"
#include "m2df_utils.h"

#include <iostream>
#include <queue>
#include <climits>

using namespace std;

namespace m2df
{
```

```

class Instance {
    unsigned *copies_cnt; //counts the copies of an instance
    unsigned graph_id, inst_id;
    unsigned root_id, graph_size; //instance id, rood node,
        graph size (expressed in nodes number)
    abstract_node *graph; //the graph

public:
    /**
     * \brief Constructor: creates a new instance, from a graph
         copy.
     * \param __gid Instance identifier.
     * \param __graph Graph structure.
     * \param __graph_size Graph structure size.
     * \param __edges Queue containing the edges (in order of
         insertion).
     * \param __root_id Position of the root node.
     */
    Instance(unsigned __gid, unsigned __iid, abstract_node *
        __graph, unsigned __graph_size, \
    queue< Edge > __edges, unsigned __root_id);
    /**
     * \brief Copy constructor.
     * \param __i Instance to copy.
     */
    Instance(const Instance& __i);
    /**
     * \brief Destructor.
     */
    virtual ~Instance();

    /**
     * \brief Assignment operator.
     * \param __i Instance to copy.
     */
    Instance& operator=(const Instance& __i);

    /**
     * \brief Getter: gets the instance id.
     */
    unsigned getGid() const {
        return graph_id;
    }
    /**
     * \brief Getter: gets the instance nodes number.
     */
    unsigned getSize() const {
        return graph_size;
    }
    /**
     * \brief Sets a task as completed. Updates the ready counts
         of all the completed
     * task's successors, and pushes the ready taska into the
```

```
         ready queue.
     * \param __tid Id of the completed task (unique for the
         instance).
     * \param __ready Reference to the ready queue.
     */
    unsigned setCompleted(unsigned __tid, queue<abstract_node>&
         __ready);
    /**
     * \brief Starts the execution of the instance. Pushes the
         root task into the ready queue.
     * \param __ready Reference to the ready queue.
     */
    void start(queue<abstract_node>& __ready);
};

} //end m2df namespace

#endif // INSTANCE_H
```

m2df_instance.cpp

```
#include "m2df_instance.h"

#include <vector>

using namespace std;

namespace m2df
{

namespace global
{
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}

Instance::Instance(unsigned __gid, unsigned __iid, abstract_node
     *__graph, unsigned __graph_size, \
    queue< Edge > __edges, unsigned __root_id)
    : graph_id(__gid), graph_size(__graph_size), root_id(
         __root_id)
{
    copies_cnt = new unsigned(1);
    unsigned *data_displ = new unsigned[__graph_size];
    unsigned *res_displ = new unsigned[__graph_size];

    graph = new abstract_node[__graph_size];
    for(unsigned i = 0; i < __graph_size; i++)
        graph[i] = __graph[i]; //new copy
    memset(&data_displ[0], 0, __graph_size * sizeof(unsigned));
    memset(&res_displ[0], 0, __graph_size * sizeof(unsigned));
```

```
    while(!__edges.empty()) {
        Edge edge = __edges.front();//.second;
        unsigned data_pos = data_displ[edge.second]++;
        unsigned res_pos = res_displ[edge.first]++;

        graph[edge.first].addr->results[res_pos] = &(graph[edge.
            second].addr->data[data_pos]);
        __edges.pop();
    }

    delete [] data_displ;
    delete [] res_displ;
}

Instance::Instance(const Instance& __i)
    : root_id(__i.root_id), graph_size(__i.graph_size),
        copies_cnt(__i.copies_cnt)
{
    (*copies_cnt)++;
    graph = __i.graph;
}

Instance::~Instance()
{
    --(*copies_cnt);
    unsigned copies_n = *copies_cnt;

    if(copies_n == 0) {
        delete copies_cnt;
        delete [] graph;
        graph = NULL;
    }
}

Instance& Instance::operator =(const Instance& __i)
{
    root_id = __i.root_id;
    graph_size = __i.graph_size;
    graph = __i.graph;

    copies_cnt = __i.copies_cnt;
    (*copies_cnt)++;
}

void Instance::start(queue<abstract_node>& __ready)
{
    __ready.push(graph[root_id]);
}

unsigned Instance::setCompleted(unsigned __tid, queue<
    abstract_node>& __ready)
{
    abstract_node completed = graph[__tid];
    register unsigned enabled = 0;
```

```
    for(unsigned j = 0; j < completed.consumers.size(); j++) {
        //decreases rdy_cnt of all consumers
        graph[completed.consumers[j]].ready_cnt--;

        if(graph[completed.consumers[j]].ready_cnt == 0) {
            register task t = *graph[completed.consumers[j]].addr;

            __ready.push(graph[completed.consumers[j]]);
            enabled++;
        }
    }

    return enabled;
}

} //end m2df namespace
```

## Interfaces

Next Listing shows the declaration of the inheritance hierarchy used by scheduer and interpreter classes.

`m2df_interfaces.h`

```
#ifndef M2DF_INTERFACES_H
#define M2DF_INTERFACES_H
#include "m2df_task.h"

namespace m2df
{
/**
 * \brief Function used for running Executable objects with pthread.
 * \param __class_ptr Pointer to the instance to be ran.
 */
void *thunk(void *__class_ptr);

/**
 * \class Runnable
 * \author Lorenzo
 * \date 10/12/2010
 * \file m2df_interfaces.h
 * \brief
 */
class Runnable {
    friend void *thunk(void*);

    /**
     * \brief Method executing the specific class behaviour. It is invoked by the thunk routine.
```

```
     */
    virtual void exec() = 0;
public:
    /**
     * \brief Method starting the class execution. It will
         create a new thread, invoking the thunk routine.
     */
    virtual void run() = 0; //TODO give a default implementation
        , in which we invoke thunk (??)
};

/**
 * \class Killable
 * \author Lorenzo
 * \date 10/12/2010
 * \file m2df_interfaces.h
 * \brief
 */
class Killable {
public:
    /**
     * \brief Method for killing the thread.
     */
    virtual void kill() = 0;
};

/**
 * \class Schedulable
 * \author Lorenzo
 * \date 10/12/2010
 * \file m2df_interfaces.h
 * \brief
 */
class Schedulable : public Runnable, public Killable {
public:
    /**
     * \brief Method for pushing a new task to be executed.
     * \param __task Task to be executed.
     * \return true, if the push completes successfully; false
         otherwise.
     */
    virtual bool pushTask(task &__task) = 0;
};

/**
 * \class Notifiable
 * \author Lorenzo
 * \date 10/12/2010
 * \file m2df_interfaces.h
 * \brief
 */
class Notifiable : public Runnable {
public:
    /**
```

```
     * \brief Method for notify the completion of a task.
     * \param __gid Instance id of the completed task.
     * \param __tid Task id of the completed task.
     */
    virtual void notifyCompletion(unsigned __gid, unsigned __tid
        ) = 0;
};

} //end m2df namespace



namespace m2df
{

inline void *thunk(void *__class_ptr) {
    Runnable *instance = (Runnable*)__class_ptr;
    instance->exec();
    return NULL;
}

} //end m2df namespace

#endif //M2DF_INTERFACES_H
```

## Interpreter

Next Listings show the interpreter class declaration and implementation.

m2df_interpreter.h

```
#ifndef INTERPRETER_H
#define INTERPRETER_H 1
#include "m2df_completion.h"
#include "m2df_debug.h"
#include "m2df_interfaces.h"
#include "m2df_queues.h"
#include "m2df_semaphore.h"
#include "m2df_task.h"
#include "m2df_utils.h"

#include <cstdlib>
#include <cstring>
#include <pthread.h>

#ifdef M2DF_PIPES_VERSION
#include <unistd.h>
#endif

#ifdef M2DF_TRACING
#include <iostream>
#include <sstream>
```

```
#endif

using namespace std;

namespace m2df
{

class Scheduler;

/**
 * \class Interpreter
 * \author Lorenzo Anardu
 * \date 10/12/2010
 * \file m2df_interpreter.h
 * \brief This class represents a MDF interpreter, running on an
      independent thread.
 */
class Interpreter : public Schedulable {
    unsigned _iid;
    bool _available, _running;
    pthread_t _thread_info;
    Notifiable *_sched_addr;

    shared_queue<task> _tasks;

    #ifdef M2DF_TRACING
    ostringstream *trace;
    unsigned elab_tasks;
    #endif

    /**
     * \brief [Private Method] Executes the interpretation loop.
     */
    void exec();
public:
    /**
     * \brief Default constructor, does nothing.
     */
    Interpreter();
    /**
     * \brief Constructor which initializes a new interpreter.
     * \param __iid Id of the interpreter.
     */
    Interpreter(unsigned __iid, Notifiable *__sched_addr);
    /**
     * \brief Destructor.
     */
    virtual ~Interpreter();

    /**
     * \brief Launches the interpreter, running on a new thread.
     */
    void run() throw(ThreadForkException,
        ThreadSchedulingException);
```

```
    /**
     * \brief Kills the interpreter thread.
     */
    void kill();
    /**
     * \brief Pushes a new task in the queue.
     * \param __task Task to be executed.
     * \return true, if the push completes successfully; false otherwise.
     */
    bool pushTask(task& __task);
};

} //end m2df namespace

#endif //INTERPRETER_H
```

### m2df_interpreter.cpp

```
#include "m2df_interpreter.h"

namespace m2df
{
namespace global
{
extern unsigned processors_number;
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}

inline Interpreter::Interpreter()
    : _iid(0), _available(true), _running(false)
{
    _sched_addr = NULL;
    _tasks.setOperation(1, false); //write operation non-blocking

    #ifdef M2DF_TRACING
    trace = new ostringstream();
    elab_tasks = 0;
    #endif
}

Interpreter::Interpreter(unsigned __iid, Notifiable * __sched_addr)
    : _iid(__iid), _available(true), _running(false)
{
    _sched_addr = __sched_addr;
    _tasks.setOperation(1, false); //write operation non-blocking

    #ifdef M2DF_TRACING
```

```
    trace = new ostringstream(ios_base::out);
    elab_tasks = 0;
    #endif

    #ifdef M2DF_DEBUG
    pthread_mutex_lock(&global::comm_mutex);
    std::cout << "Interpreter␣" << _iid << "␣created␣succesfully." << std::endl;
    std::cout.flush();
    pthread_mutex_unlock(&global::comm_mutex);
    #endif
}

Interpreter::~Interpreter()
{
}

void Interpreter::run() throw(ThreadForkException, ThreadSchedulingException)
{
    int error_code;
    if(_running == true) return; //it is not possible launching 2 times the same interpreter

    if((error_code = pthread_create(&_thread_info, NULL, thunk, (void*)this)) != 0) //some error occurred
        throw ThreadForkException();//(error_code);
    else _running = true;
}

inline void Interpreter::kill()
{
    #ifdef M2DF_TRACING
    (*trace)<<_iid<<"␣ELABORATED␣"<<elab_tasks<<"␣TASKS\n";
    #endif
    pthread_cancel(_thread_info);
}

bool Interpreter::pushTask(task& __task)
{
        if(_available == true) {
        try {
            _tasks.push(__task);
        } catch(FullQueueException e) {
            cout << "Interpreter␣"<<_iid<<"␣fullqueue."<<endl;
            _available = false;
            return false;
        }
        return true;
    }
    else return false;
}

void Interpreter::exec()
```

```
{
    #ifdef M2DF_DEBUG
    pthread_mutex_lock(&global::comm_mutex);
    std::cout << "Interpreter␣" << _iid << "␣running." << std::
        endl;
    std::cout.flush();
    pthread_mutex_unlock(&global::comm_mutex);
    #endif
    int error_code;
    cpu_set_t cpu_set;
    unsigned cpu_num = global::processors_number;

    CPU_ZERO(&cpu_set);
    if(_iid < cpu_num)
        CPU_SET(_iid % cpu_num, &cpu_set);
    else for(unsigned i = 0; i < cpu_num; i++)
            CPU_SET(i, &cpu_set); //excess parallelism
                interpreters can run in any cpu

    if((error_code = pthread_setaffinity_np(_thread_info, sizeof
        (cpu_set_t), &cpu_set)) != 0) //some error occurred
        throw ThreadSchedulingException(error_code);

    if(pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL)
        != 0) {
        cerr<<"Pthread␣setcanceltype␣error"<<endl;
    }

    while(true) { //interpreter is killed by the scheduler
        register task t = _tasks.pop();

        for(unsigned i = 0; i < t.data_n; i++)
            __builtin_prefetch(t.data[i], 0, 3); //transfer into
                L1 cache

        void **res = t.code(t.data);

        for(unsigned i = 0; i < t.res_n; i++) {
            memcpy(t.results[i], &res[i], sizeof(void*));
        }
        _sched_addr->notifyCompletion(t.gid, t.tid);
        _available = true;

        #ifdef M2DF_DEBUG
        pthread_mutex_lock(&global::comm_mutex);
        std::cout << "Interpreter␣" << _iid << ":␣Task␣<" << t.
            gid << ",␣" << t.tid << ">␣completed." << std::endl;
        std::cout.flush();
        pthread_mutex_unlock(&global::comm_mutex);
        #endif

        #ifdef M2DF_TRACING
        elab_tasks++;
        (*trace)<<_iid<<"\t"<<t.gid<<"\t"<<t.tid<<endl;
```

```
        #endif
    }
}
}
```

## Parameters

Next Listings show the declaration and implementation of the classes used to manage the input and output of data from the task routines.

`m2df_parameters.h`

```
#ifndef PARAMETERS_H
#define PARAMETERS_H 1
#include "m2df_exceptions.h"

#include <cstdlib>

using namespace std;

namespace m2df
{
/**
 * \class ParameterBase
 * \author Lorenzo
 * \date 10/01/2011
 * \file m2df_parameters.h
 * \brief Base class for parameters management.
 */
class ParameterBase {
protected:
    void **_buf;
public:
    /**
     * \brief Creates an empty parameter.
     */
    ParameterBase() : _buf(NULL) {}
    /**
     * \brief Creates a parameter, allocating the space for n
         tuples.
     * \param n_elms Number of tuples composing the parameter.
     */
    ParameterBase(unsigned n_elms);
    /**
     * \brief Creates a parameter by a pre-allocated buffer.
     * \param buf Buffer containing the parameter data.
     */
    ParameterBase(void **buf) : _buf(buf) {}
    /**
     * \brief Destructor: Does nothing.
     */
    virtual ~ParameterBase() {}
```

```

    /**
     * \brief Operator (): simulates the construction of an object.
     * \param new_n Number of positions in the buffer.
     */
    void operator()(unsigned new_n);
    /**
     * \brief Operator (): simulates the construction of an object.
     * \param newbuf Buffer containing the parameter data.
     */
    void operator()(void **newbuf);

    /**
     * \brief Allocates the space for n tuples.
     * \param n_elms Number of tuples composing the parameter.
     */
    void allocate(unsigned n_elms);
    /**
     * \brief Deallocates the parameter.
     */
    void deallocate();

    /**
     * \brief Returns a pointer to the space allocated.
     * \return A pointer to the allocated buffer.
     */
    void **get();
};

/**
 * \class Parameter
 * \author Lorenzo
 * \date 10/01/2011
 * \file m2df_parameters.h
 * \brief Class representing a parameter. A parameter is an array of Tuples.
 */
class Parameter : public ParameterBase {
public:
    Parameter() : ParameterBase() {}
    /**
     * \brief Creates a parameter, allocating the space for n tuples.
     * \param n_elms Number of tuples composing the parameter.
     */
    Parameter(unsigned n_elms) : ParameterBase(n_elms) {}
    /**
     * \brief Creates a parameter by a pre-allocated buffer.
     * \param buf Buffer containing the parameter data.
     */
    Parameter(void **buf) : ParameterBase(buf) {}

```

```
    /**
     * \brief Sets the n^th tuple.
     * \param at Position to be set.
     * \param tuple Buffer of the tuple to be set.
     */
    void setTupleAt(unsigned at, void **tuple);
    /**
     * \brief Gets the n^th tuple's buffer.
     * \param at Position to be get.
     */
    void **getTupleAt(unsigned at);
};

/**
 * \class Tuple
 * \author Lorenzo
 * \date 10/01/2011
 * \file m2df_parameters.h
 * \brief Class representing a Tuple. A Tuple is an array of
     Elements.
 */
class Tuple : public ParameterBase {
public:
    /**
     * \brief Creates a tuple, allocating the space for n
         elements.
     * \param n_elms Number of elements composing the parameter.
     */
    Tuple(unsigned n_elms) : ParameterBase(n_elms) {}
    /**
     * \brief Creates a tuple by a pre-allocated buffer.
     * \param buf Buffer containing the parameter data.
     */
    Tuple(void **buf) : ParameterBase(buf) {}

    /**
     * \brief Sets the n^th element.
     * \param at Position to be set.
     * \param elm Address to the elment to be set.
     */
    void setElementAt(unsigned at, void *elm);
    /**
     * \brief Gets the n^th tuple's element.
     * \param at Position to be get.
     */
    void *getElementAt(unsigned at);
};


template<class T>
/**
 * \class parameter_cast
 * \author Lorenzo
 * \date 10/01/2011
```

```
 * \file m2df_parameters.h
 * \brief Class for easy and safe casting of parameters to/from void*
 * Specific behaviour for base types. A new element is allocated and
 * its address is converted to void*. Inverse operation consists in
 * a pointer dereferentiation.
 */
class parameter_cast {
public:
/**
 * \brief Casts an element to void* (different behaviour for variables and pointers)
 * \param elm Element to be casted.
 * \return Address of a copy of the element, casted to void*.
 */
static void *cast(T elm) {
    return new T(elm);
}
/**
 * \brief Casts an element from void* (different behaviour for variables and pointers)
 * \param elm Address of the element to be casted.
 * \return Element.
 */
static T cast(void *elm) {
    return (*(T*)elm);
}
};

template<class T>
/**
 * \class parameter_cast<T*>
 * \author Lorenzo
 * \date 10/01/2011
 * \file m2df_parameters.h
 * \brief Class for easy and safe casting of parameters to/from void*
 * Specific behaviour for pointers to base types. The element's address
 * is converted to void*. Inverse operation consists in a simple cast.
 */
class parameter_cast<T*> {
public:
/**
 * \brief Casts a pointer to void* (different behaviour for variables and pointers)
 * \param elm Address of the element to be casted.
 * \return Address, casted to void*.
 */
static void *cast(T *elm) {
    return elm;
```

```
}
/**
 * \brief Casts an element from void* (different behaviour for variables and pointers)
 * \param elm Address of the element to be casted.
 * \return Address, casted to T*.
 */
static T* cast(void *elm) {
    return ((T*)elm);
}
};

} //end namespace m2df

#endif //PARAMETERS_H
```

m2df_parameters.cpp

```
#include "m2df_parameters.h"

namespace m2df
{
ParameterBase::ParameterBase(unsigned n_elms)
{
    _buf = (void**)calloc(n_elms, sizeof(void*));
    if(_buf == NULL) throw BadAllocException();
}

void ParameterBase::operator()(unsigned new_n)
{
    this->allocate(new_n);
}

void ParameterBase::operator()(void **newbuf)
{
    _buf = newbuf;
}

void ParameterBase::allocate(unsigned n_elms)
{
    _buf = (void**)calloc(n_elms, sizeof(void*));
    if(_buf == NULL) throw BadAllocException();
}

void ParameterBase::deallocate()
{
    if(_buf != NULL) {
        free(_buf);
        _buf = NULL;
    }
}

void **ParameterBase::get()
```

```
{
    return _buf;
}


void Parameter::setTupleAt(unsigned at, void **tuple)
{
    if(_buf != NULL)
        _buf[at] = tuple;
}

void **Parameter::getTupleAt(unsigned at)
{
    void **out = NULL;

    if(_buf != NULL)
        out = (void**)_buf[at];

    return out;
}


void Tuple::setElementAt(unsigned at, void *elm)
{
    if(_buf != NULL)
        _buf[at] = elm;
}

void *Tuple::getElementAt(unsigned at)
{
    void **out = NULL;

    if(_buf != NULL)
        out = (void**)_buf[at];

    return out;
}

} //end namespace m2df
```

## Queue

Next Listing shows the implementation of the shared queue class. Since it is a template class, the implementation resides in the same `.h` file in which it is declared.

`m2df_queues.h`

```
#ifndef QUEUES_H
#define QUEUES_H 1
#include "m2df_completion.h"
#include "m2df_debug.h"
#include "m2df_semaphore.h"
#include "m2df_task.h"

#include <assert.h>
#include <cstdlib>
#include <fcntl.h>
#include <poll.h>
#include <pthread.h>
#include <queue>
#include <unistd.h>

using namespace std;

namespace m2df
{
/**
 * \class shared_queue
 * \author Lorenzo Anardu
 * \date 10/12/2010
 * \file m2df_queues.h
 * \brief A templated thread safe queue.
 */
template <class T>
class shared_queue
{
    bool _w_block, _r_block;
    #ifdef M2DF_SYNC_VERSION
    pthread_mutex_t _mutex;
    Semaphore _entries_n;
    queue<T> _data;
    #endif

    #ifdef M2DF_PIPES_VERSION
    int _pipe_des[2];
    #endif

public:
    shared_queue();
    ~shared_queue();

    /**
     * \brief Sets the value of the blocking flag for the
```

```
            specified operation.
     * \param __op Operation to be set. (0=read, 1=write)
     * \param __blocking Flag value. (true=blocking, false=non-
            blocking) [Default val=true]
     */
    void setOperation(short __op, bool __blocking);

    /**
     * \brief Adds an element in the bottom of the queue.
            Suspends on full queue.
     * \param __data Element to be added.
     */
    inline void push(T& __data);
    /**
     * \brief Removes an element from the front of the queue.
            Suspends on empty queue.
     * \return The element that has been removed.
     */
    inline T pop();
    /**
     * \brief Removes an element from the front of the queue, if
             present. If the queue is empty throws an
            EmptyQueueException.
     * \return The element that has been removed.
     */
    inline T tryPop();
    /**
     * \brief Removes an element from the front of the queue, if
             present. Waits at max time milliseconds if the queue is
             empty.
     * \param time Time to wait, in milliseconds.
     * \param valid Output parameter. Indicates whether the
            returned T is valid or not.
     * \return The element that has been removed.
     */
     inline T timedPop(unsigned time, bool *valid);
    /**
     * \brief Indicates whether the queue is empty.
     * \return true if the queue is empty, false otherwise.
     */
    bool isEmpty();
    /**
     * \brief Indicates whether the queue is full.
     * \return true if the queue is full, false otherwise.
     */
    bool isFull();
};

} //end m2df namespace



namespace m2df
{
```

```
template <class T>
inline shared_queue<T>::shared_queue() : _w_block(true),
    _r_block(true)
{
    #ifdef M2DF_SYNC_VERSION
    pthread_mutex_init(&_mutex, NULL);
    _entries_n = Semaphore(0, ULONG_MAX);
    #endif

    #ifdef M2DF_PIPES_VERSION
    if((pipe(_pipe_des)) == -1) {
        throw PipeException();
    }
    #endif
}

template <class T>
shared_queue<T>::~shared_queue()
{
    #ifdef M2DF_SYNC_VERSION
    pthread_mutex_destroy(&_mutex);
    #endif

    #ifdef M2DF_PIPES_VERSION
    close(_pipe_des[1]); //write-end
    close(_pipe_des[0]); //read-end
    #endif
}

template <class T>
void shared_queue<T>::setOperation(short __op, bool __blocking)
    {
    if(__op > 1) return;

    if(__op == 0) _r_block = __blocking;
    if(__op == 1) _w_block = __blocking;

    #ifdef M2DF_PIPES_VERSION
    int flags = fcntl(_pipe_des[__op], F_GETFL, 0);
    assert(flags != -1);

    if(__blocking) fcntl(_pipe_des[__op], F_SETFL, flags |
        O_NONBLOCK); // set operation non-blocking
    else fcntl(_pipe_des[__op], F_SETFL, flags & ~O_NONBLOCK);
        // set operation blocking
    #endif
}

template <class T>
inline void shared_queue<T>::push(T& __data) {
    #ifdef M2DF_SYNC_VERSION
    if(!_w_block && !_entries_n.isPostable())
        throw FullQueueException(); //throw exception only if
            non-blocking
```

```
    pthread_mutex_lock(&_mutex);
    _data.push(__data);
    pthread_mutex_unlock(&_mutex);
    _entries_n.post();  //ATTENTION, possibility of race-
        condition
    #endif

    #ifdef M2DF_PIPES_VERSION
    T* temp=&__data;
    if(write(_pipe_des[1], &temp, sizeof(T*)) == -1) { //writing
         error
        if(!_w_block) throw FullQueueException(); //throw
            exception only if non-blocking

        cerr<<"Error␣in␣writing␣into␣pipe:␣"<<strerror(errno)<<
            endl;
        cout.flush();
        throw PipeWriteException();
    }
    #endif
}

template <class T>
inline T shared_queue<T>::pop() {
    #ifdef M2DF_SYNC_VERSION
    T ret;

    if(!_r_block && !_entries_n.isWaitable())
        throw EmptyQueueException(); //throw exception only if
            non-blocking

    _entries_n.wait(); //thread is suspended until n == 0
    pthread_mutex_lock(&_mutex);
    ret = _data.front();
    _data.pop();
    pthread_mutex_unlock(&_mutex);

    return ret;
    #endif

    #ifdef M2DF_PIPES_VERSION
    T *ret;
    if(read(_pipe_des[0], &ret, sizeof(T*)) == -1) { //reading
        error
        if(!_r_block) throw EmptyQueueException(); //throw
            exception only if non-blocking

        cerr<<"Error␣in␣reading␣from␣pipe:␣"<<strerror(errno)<<
            endl;
        cout.flush();
        throw PipeReadException();
    }

```

```
    return (*ret);
    #endif
}

template <class T>
inline T shared_queue<T>::tryPop() {
    #ifdef M2DF_SYNC_VERSION
    T ret;

    pthread_mutex_lock(&_mutex);
    if(!_entries_n.isWaitable()) {
        pthread_mutex_unlock(&_mutex);
        throw EmptyQueueException(); //throw exception only if non-blocking
    }

    _entries_n.wait(); //thread is suspended until n == 0
    ret = _data.front();
    _data.pop();
    pthread_mutex_unlock(&_mutex);

    return ret;
    #endif

    #ifdef M2DF_PIPES_VERSION
    T *ret;
    if(read(_pipe_des[0], &ret, sizeof(T*)) == -1) { //reading error -> empty pipe
        throw EmptyQueueException();
    }

    return (*ret);
    #endif
}

template <class T>
inline T shared_queue<T>::timedPop(unsigned time, bool *valid) {
    #ifdef M2DF_SYNC_VERSION
    T ret;

    pthread_mutex_lock(&_mutex);
    *valid = _entries_n.timedWait(time); //thread is suspended until n == 0
    if(*valid) {
        ret = _data.front();
        _data.pop();
    }
    else ret = T(); //timeout
    pthread_mutex_unlock(&_mutex);

    return ret;
    #endif

    #ifdef M2DF_PIPES_VERSION
```

```
    struct pollfd p;
    int ret;

    p.fd = _pipe_des[0];
    p.events = POLLIN; //pipe is empty if it is not readable

    if((ret = poll(&p, 1, time)) == -1) {
        cerr<<"Error in polling the pipe: "<<strerror(errno)<<
            endl;
        cout.flush();
        throw PipeWriteException();
    }

    if(ret > 0) {
        T *ret;
        if(read(_pipe_des[0], &ret, sizeof(T*)) == -1) { //
            reading error -> empty pipe
            throw EmptyQueueException();
        }
        *valid = true;
        return (*ret);
    }
    else {
        *valid = false;
        return T();
    }
    #endif
}

template <class T>
bool shared_queue<T>::isEmpty() {
    #ifdef M2DF_SYNC_VERSION
    return !_entries_n.isWaitable();
    #endif

    #ifdef M2DF_PIPES_VERSION
    struct pollfd p;
    int ret;

    p.fd = _pipe_des[0];
    p.events = POLLIN; //pipe is empty if it is not readable

    if((ret = poll(&p, 1, 0)) == -1) {
        cerr<<"Error in polling the pipe: "<<strerror(errno)<<
            endl;
        cout.flush();
        throw PipeWriteException();
    }

    return (ret == 0);
    #endif
}

template <class T>
```

```
bool shared_queue<T>::isFull()
{
    #ifdef M2DF_SYNC_VERSION
    return !_entries_n.isPostable();
    #endif

    #ifdef M2DF_PIPES_VERSION
    struct pollfd p;
    int ret;

    p.fd = _pipe_des[1];
    p.events = POLLOUT; //pipe is full if it is not writable

    if((ret = poll(&p, 1, 0)) == -1) {
        cerr<<"Error␣in␣polling␣the␣pipe:␣"<<strerror(errno)<<
            endl;
        cout.flush();
        throw PipeWriteException();
    }

    return (ret == 0);
    #endif
}

} //end m2df namespace

#endif //QUEUES_H
```

## Schedule

Next Listing shows the implementation of the scheduling policy. Since this function was declared to be `inline`, its implementation resedes in the same `.h` file in which it is declared.

In case someone wants to modify the scheduling policy this *should* be the only modification point.

`m2df_schedule.h`

```
#ifndef SCHEDULE_H
#define SCHEDULE_H 1
#include "m2df_abstract_node.h"
#include "m2df_debug.h"
#include "m2df_interfaces.h"
#include "m2df_priority.h"
#include "m2df_task.h"

#include <iostream>
#include <queue>
#include <vector>
using namespace std;

namespace m2df
{
namespace global
{
#ifdef M2DF_DEBUG
extern pthread_mutex_t comm_mutex;
#endif
}

/**
 * \brief Scheduling function. It implements the scheduling
     pilicy used by the Scheduler.
 * Changes to the scheduling policy must be done HERE.
 * \param __tasks Set of tasks ready to be launched.
 * \param __procs Set of processors(interpreters) for executing
     the tasks.
 */
inline bool schedule(queue<abstract_node>& __tasks, vector<
    Schedulable*>& __procs){
    if(__tasks.empty()) return false;
    task *t = __tasks.front().addr;

    // If u want to change the policy
    // comment from here...
    static unsigned i = 0;
    bool scheduled = false;

    scheduled = __procs[i]->pushTask(*t);
    if(scheduled) {
        #ifdef M2DF_DEBUG
```

```
            pthread_mutex_lock(&global::comm_mutex);
            std::cout << "Task␣<" << t->gid << ",␣" << t->tid <<
                ">␣sent␣to␣" << i <<std::endl;
            std::cout.flush();
            pthread_mutex_unlock(&global::comm_mutex);
        #endif
        __tasks.pop();
        i = (i + 1) % __procs.size();
        return true;
    }
    else return false;
    //...until here (or sort it out yourself!)

    return scheduled;
}

} //end m2df namespace

#endif //SCHEDULE_H
```

## Scheduler

Next Listings show the declaration and implementation of the scheduler class.

m2df_scheduler.h

```
#ifndef SCHEDULER_H
#define SCHEDULER_H 1
#include "m2df_abstract_node.h"
#include "m2df_completion.h"
#include "m2df_debug.h"
#include "m2df_instance.h"
#include "m2df_interfaces.h"
#include "m2df_interpreter.h"
#include "m2df_priority.h"
#include "m2df_schedule.h"
#include "m2df_semaphore.h"

#include <assert.h>
#include <cstdlib>
#include <fcntl.h>
#include <map>
#include <pthread.h>
#include <queue>
#include <signal.h>
#include <unistd.h>
#include <utility>
#include <vector>

#ifdef M2DF_TRACING
#include <fstream>
```

```
#include <iostream>
#include <string>
#endif

using namespace std;

namespace m2df
{
class Graph;

/**
 * \class Scheduler
 * \author Lorenzo Anardu
 * \date 10/12/2010
 * \file m2df_scheduler.h
 * \brief This class represents a MDF scheduler, running on the
     master thread.
 */
class Scheduler : public Notifiable {
    bool _enabled_compl;
    unsigned _tasks_n, _int_n;

    pthread_mutex_t _mutex; //used for safe pushing of new
        graphs
    pthread_cond_t _block; //used for blocking when completion
        is not enabled and we have to wait for new instances
    pthread_t _thread_info;

    vector< Instance > _graphs;
    queue< abstract_node > _ready_tasks;
    vector< Schedulable* > _procs;
    shared_queue< completion > _completed_tasks;


    void addInstance(Instance __inst) throw(BadAllocException);
    void exec();

    friend class Graph;
public:
    /**
     * \brief Constructor creating a scheduler which runs on a
         multicore machine.
     * \param __cores_n Number of cores available in the machine
         .
     */
    Scheduler(unsigned __interp_n);
    /**
     * \brief Destructor.
     */
    virtual ~Scheduler();

    /**
     * \brief Executes the scheduler loop.
     */
```

```
    void run() throw(ThreadForkException,
        ThreadSchedulingException);
    /**
     * \brief Gets the number of interpreters that actually
         running.
     * \return The number of interpreters.
     */
    const unsigned GetInterpretersNumber() const {
        return _procs.size();
    }
    /**
     * \brief Gets the thread info of the scheduler.
     * \return The thread informetion of the scheduler.
     */
    const pthread_t GetThreadInfo() const {
        return _thread_info;
    }

    /**
     * \brief Enables the scheduler to terminate. The scheduler
         terminates iff
     * the completion has been enabled AND all the submitted
         tasks have been computed.
     */
    void enableCompletion();
    /**
     * \brief Notifies to the scheduler the completion of a task
         .
     * \param __gid Graph id of the completed task.
     * \param __tid Task id of the task.
     */
    void notifyCompletion(unsigned __gid, unsigned __tid);
};

} //end m2df namespace

#endif //SCHEDULER_H
```

m2df_scheduler.cpp

```
#include "m2df_scheduler.h"

#include <sys/time.h>
#include <iomanip>

#define MIN(X, Y) (X < Y)? X : Y

namespace m2df
{

namespace global
{
extern unsigned processors_number;
```

```
#ifdef M2DF_DEBUG
extern pthread_mutex_t comm_mutex;
#endif
}

Scheduler::Scheduler(unsigned __interp_n)
    : _enabled_compl(false), _tasks_n(0)
{
    #ifdef M2DF_SYNC_VERSION
    cout << "|=========|SYNC_VERSION|=========|"<<endl;
    #endif
    #ifdef M2DF_PIPES_VERSION
    cout << "|=========|PIPES_VERSION|=========|"<<endl;
    #endif

    for(unsigned i = 0; i < __interp_n; i++) //creates interp_n interpreters, but doesn't launch them
        _procs.push_back(new Interpreter(i, this));

    pthread_mutex_init(&_mutex, NULL);
    pthread_cond_init(&_block, NULL);

    #undef M2DF_DEBUG
    #ifdef M2DF_DEBUG
    pthread_mutex_lock(&global::comm_mutex);
    cout << "Scheduler created succesfully." << endl;
    std::cout.flush();
    pthread_mutex_unlock(&global::comm_mutex);
    #endif
}

Scheduler::~Scheduler()
{
    _procs.clear();
    _graphs.clear();
    pthread_mutex_destroy(&_mutex);
    pthread_cond_destroy(&_block);
}

void Scheduler::addInstance(Instance __inst) throw(BadAllocException)
{
    pthread_mutex_lock(&_mutex);
    _graphs.push_back(__inst);
    __inst.start(_ready_tasks);

    _tasks_n += __inst.getSize();

    pthread_cond_signal(&_block);
    pthread_mutex_unlock(&_mutex);
}

void Scheduler::enableCompletion() //called by the user thread (finalize)
```

```
{
    pthread_mutex_lock(&_mutex);
    _enabled_compl = true;
    pthread_cond_signal(&_block);
    pthread_mutex_unlock(&_mutex);
}

inline void Scheduler::notifyCompletion(unsigned __gid, unsigned __tid)
{
    completion *compl_task = new completion(__gid, __tid);

    _completed_tasks.push(*compl_task);
}

void Scheduler::run() throw(ThreadForkException, ThreadSchedulingException)
{
    struct sched_param my_param;
    pthread_attr_t my_attr;
    int error_code;

//set high scheduling priority, IS IT EFFECTIVE?
    pthread_attr_init(&my_attr);
    pthread_attr_setinheritsched(&my_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&my_attr, SCHED_OTHER);
    my_param.__sched_priority = sched_get_priority_max(SCHED_OTHER);
    pthread_attr_setschedparam(&my_attr, &my_param);

    if((error_code = pthread_create(&_thread_info, &my_attr, thunk, (void*)this)) != 0)
        throw ThreadForkException(error_code);

    unsigned cpu_number = MIN(global::processors_number, _procs.size());
    cpu_set_t cpu_set;

    CPU_ZERO(&cpu_set);
    for(unsigned cpu_id = 0; cpu_id < cpu_number; cpu_id++)
        CPU_SET(cpu_id, &cpu_set); //scheduler can run in any (assigned) core

    if((error_code = pthread_setaffinity_np(_thread_info, sizeof(cpu_set_t), &cpu_set)) != 0) //some error occurred
        throw ThreadSchedulingException(error_code);


    #ifdef M2DF_DEBUG
    if((error_code = pthread_getaffinity_np(_thread_info, sizeof(cpu_set_t), &cpu_set)) != 0)
        throw ThreadSchedulingException(error_code);
```

```
    pthread_mutex_lock(&global::comm_mutex);
    cout << "Scheduler␣'s␣mask:␣";
    for (unsigned j = 0; j < CPU_SETSIZE; j++)
        if (CPU_ISSET(j, &cpu_set)) cout<<j<<"␣␣";
    cout << endl;
    cout.flush();
    pthread_mutex_unlock(&global::comm_mutex);
    #endif
}

void Scheduler::exec()
{
    unsigned compl_tasks = 0;

    cout << "Scheduler␣running␣with␣" << _tasks_n << "␣tasks,␣and␣" << _procs.size() << "␣interpreters." << endl;
    std::cout.flush();

    for(unsigned i = 0; i < _procs.size(); i++)
        _procs[i]->run();

    while(true) {
        timeval start_t, end_t;
        double cur_t;
        pthread_mutex_lock(&_mutex);

        register unsigned enabled_tasks = 0;
        if(_enabled_compl && compl_tasks == _tasks_n) break; // finished!
        else if(compl_tasks == _tasks_n) { //wait for new instances
            pthread_cond_wait(&_block, &_mutex);
            if(_enabled_compl && compl_tasks == _tasks_n) break; // finished!
            //otherwise further instances were submitted
        }

        while(schedule(_ready_tasks, _procs)) ; //schedule until there is something to schedule

        pthread_mutex_unlock(&_mutex);

        #ifdef M2DF_DEBUG
        pthread_mutex_lock(&global::comm_mutex);
        std::cout << "Some␣task␣completed" << std::endl;
        std::cout.flush();
        pthread_mutex_unlock(&global::comm_mutex);
        #endif

        register bool cond;
        do {
            completion compl_task = _completed_tasks.pop();
            unsigned c_tid = compl_task.tid;
            unsigned c_gid = compl_task.gid;
```

```
            pthread_mutex_lock(&_mutex);

            enabled_tasks = _graphs[compl_task.gid].setCompleted(c_tid, _ready_tasks);

            compl_tasks++;
            if(enabled_tasks) {
                while(schedule(_ready_tasks, _procs)) ;
                enabled_tasks = 0;
            }

            cond = (compl_tasks < _tasks_n);

            pthread_mutex_unlock(&_mutex);
        } while(cond);
    }

    pthread_mutex_unlock(&_mutex);

    for(unsigned i = 0; i < _procs.size(); i++)
        _procs[i]->kill();

    #ifdef M2DF_TRACING
    std::fstream trace_file(”/tmp/tracing_info”, std::fstream::out | fstream::binary);
    if(!trace_file.is_open() || trace_file.eof() || !trace_file.good()) pthread_exit(&compl_tasks);

    for(unsigned i = 0; i < procs.size(); i++) {
        std::string str = procs[i]->trace->str();
        trace_file.write(str.c_str(), str.size());
    }
    trace_file.close();
    #endif

    pthread_exit(&compl_tasks);
}

} //end m2df namespace
```

## Semaphore

Next Listings show the declaration and implementation of the semaphore class.

`m2df_semaphore.h`

```
#ifndef SEMAPHORE_H
#define SEMAPHORE_H 1
#include "m2df_debug.h"

#include <cerrno>
#include <climits>
#include <ctime>
#include <pthread.h>
#include <sys/time.h>

namespace m2df
{

/**
 * \class Semaphore
 * \author Lorenzo Anardu
 * \date 10/12/2010
 * \file m2df_semaphore.h
 * \brief Semaphore implementation, based on pthread_cond and
     pthread_mutex.
 */
class Semaphore {
    unsigned long val, max;
    bool suspended;
    pthread_mutex_t mutex;
    pthread_cond_t cond;

public:
    /**
     * \brief Constructor which initializes the semaphore to 0.
     */
    Semaphore();
    /**
     * \brief Constructor which initializes the semaphore to a
         value different than 0.
     * \param __val Value used for initializing the semaphore.
     * \param __max [Default value = INT_MAX]Maximum value that
         the semaphore can reach.
     */
    Semaphore(unsigned long __val, unsigned long __max =
        ULONG_MAX);
    /**
     * \brief Destructor.
     */
    ~Semaphore();
    /**
     * \brief Blocking wait operation.
```

```
     */
    void wait();
    /**
     * \brief Non-blocking wait operation.
     * \return Returns whether the wait was successful.
     */
    bool tryWait();
    /**
     * \brief Non-blocking wait operation.
     * \param n Value to be decreased from the semaphore.
     * \return Returns whether the wait was successful.
     */
    bool tryWait(unsigned long n);
    /**
     * \brief Timed wait operation.
     * \param time Time to wait in milliseconds. If time = -1 waits for ever.
     * \return Returns whether the wait was successful.
     */
    bool timedWait(int w_time);
    /**
     * \brief Post operation. Wakes up a suspended thread (if any).
     */
    bool post();
    /**
     * \brief Gets the value of the semaphore.
     * \return The value of the semaphore.
     */
    unsigned long getValue();
    /**
     * \brief Gets the maximum value of the semaphore.
     * \return The maximum value of the semaphore.
     */
    const long getMax();
    /**
     * \brief Indicates whether the semaphore is postable.
     * \return A value indicating whether the semaphore is postable.
     */
    bool isPostable();
    /**
     * \brief Indicates whether the semaphore is waitable.
     * \return A value indicating whether the semaphore is waitable.
     */
    bool isWaitable();
    bool someWaiting();
};

} //end m2df namespace

#endif //SEMAPHORE_H
```

m2df_semaphore.cpp

```cpp
#include "m2df_semaphore.h"

namespace m2df
{

namespace global
{
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}

Semaphore::Semaphore()
    : val(0), max(ULONG_MAX), suspended(false)
{
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
}

Semaphore::Semaphore(unsigned long __val, unsigned long __max)
    : val(__val), max(__max), suspended(false)
{
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
}

Semaphore::~Semaphore()
{
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
}

unsigned long Semaphore::getValue()
{
    unsigned long v;
    pthread_mutex_lock(&mutex);
    v = val;
    pthread_mutex_unlock(&mutex);
    return v;
}

bool Semaphore::post()
{
    bool success = true;
    pthread_mutex_lock(&mutex);
    if(val < max) val++;
    else success = false;
    if(val == 1 && success == true) {
        pthread_cond_signal(&cond);
    }
    pthread_mutex_unlock(&mutex);
    return success;
```

```
}

bool Semaphore::tryWait()
{
    bool ret = true;
    pthread_mutex_lock(&mutex);
    if(val > 0) val--;
    else ret = false;
    pthread_mutex_unlock(&mutex);
    return ret;
}

bool Semaphore::tryWait(unsigned long n)
{
    bool ret = true;
    if(n == 0) return false;  //throw Exception ?
    pthread_mutex_lock(&mutex);
    if(val > n) val -= n;
    else ret = false;
    pthread_mutex_unlock(&mutex);
    return ret;
}

bool Semaphore::timedWait(int w_time)
{
    bool ret = true;
    timespec t = {0, 0};
    timeval actual;

    if(w_time == -1) { //waits for ever
        wait();
        return true;
    }

    pthread_mutex_lock(&mutex);
    if(val == 0) {
        gettimeofday(&actual, NULL);

        t.tv_sec = actual.tv_sec + (w_time / 1000);
        t.tv_nsec = actual.tv_usec * 1000 + (unsigned long)(
            w_time % 1000) * 1000000;

        int result = pthread_cond_timedwait(&cond, &mutex, &t);
        if( result == ETIMEDOUT ) {
            pthread_mutex_unlock(&mutex);
            return false;
        }
    }
    val--;
    pthread_mutex_unlock(&mutex);
    return true;
}

void Semaphore::wait()
```

```
{
    pthread_mutex_lock(&mutex);
    if(val == 0)
        pthread_cond_wait(&cond, &mutex);
    val--;
    pthread_mutex_unlock(&mutex);
}

const long Semaphore::getMax()
{
    return max;
}

bool Semaphore::isPostable()
{
    bool ret;
    pthread_mutex_lock(&mutex);
    if(val < max) ret = true;
    else ret = false;
    pthread_mutex_unlock(&mutex);
    return ret;
}

bool Semaphore::isWaitable()
{
    bool ret;
    pthread_mutex_lock(&mutex);
    ret = (val > 0);
    pthread_mutex_unlock(&mutex);
    return ret;
}

bool Semaphore::someWaiting()
{
    bool ret;
    pthread_mutex_lock(&mutex);
    ret = suspended;
    pthread_mutex_unlock(&mutex);
    return ret;
}

} //end m2df namespace
```

## Task

Next Listings show the declaration and implementation of the concrete task representation.

`m2df_task.h`

```
#ifndef TASK_H
#define TASK_H 1
#include "m2df_debug.h"
#include "m2df_exceptions.h"

#include <cstdlib>
#include <cstring>
#include <pthread.h>

namespace m2df
{

/**
 * \struct _task
 * \author Lorenzo Anardu
 * \date 09/12/2010
 * \file m2df_task.h
 * \brief Concrete representation of a MDF task.
 */
struct _task {
private:
    unsigned *copies_cnt; //counts the copies of an instance
    pthread_mutex_t _mutex;
public:
    unsigned gid, tid, uid; //graph_id, instance_id, task_id,
        unique_id
    unsigned data_n, res_n; //size of data and results arrays
    void **(*code)(void**); //pointer to the routine
    void **data, **results; //data and results arrays

    /**
     * \brief Default constructor, does nothing.
     */
    _task();
    /**
     * \brief Creates the task, and allocates the resources.
     * \param __graph_id Id of the reference graph.
     * \param __inst_id Id of the instance of reference graph.
     * \param __task_id Id of the node within the graph.
     * \param __unique_id Unique id of the node.
     * \param __func Code to be executed.
     * \param __data_n Number of input data required.
     * \param __res_n Number of results produced.
     */
    _task(unsigned __graph_id, unsigned __task_id, unsigned
        __unique_id, \
        void **(*__func)(void**), unsigned __data_n, unsigned
```

```
            __res_n, void **data = NULL) throw(BadAllocException)
            ;
    /**
     * \brief Copy constructor.
     * \param __t Task to be copied.
     */
    _task(const _task& __t) throw(BadAllocException);
    /**
     * \brief Destructor.
     */
    virtual ~_task();

    /**
     * \brief Assignment operator.
     * \param __t Task to copy.
     */
    _task& operator=(const _task& __t) throw(BadAllocException);
};

typedef struct _task task;

} //end m2df namespace



namespace m2df
{
inline _task::_task()
    : gid(0), tid(0), uid(0), data_n(0), res_n(0), code(NULL),
        data(NULL), results(NULL)
{
}

} //end m2df namespace

#endif //TASK_H
```

m2df_task.cpp

```
#include "m2df_task.h"

using namespace std;

namespace m2df
{
namespace global
{
#ifdef M2DF_DEBUG
    extern pthread_mutex_t comm_mutex;
#endif
}

_task::_task(unsigned __graph_id, unsigned __task_id, unsigned
    __unique_id, \
```

```
              void **(*__func)(void**), unsigned __data_n,
                  unsigned __res_n, void **__data) throw(
                  BadAllocException)
    : gid(__graph_id), tid(__task_id), uid(__unique_id), \
      code(__func), data_n(__data_n), res_n(__res_n)
{
    pthread_mutex_init(&_mutex, NULL);
    copies_cnt = new unsigned(1);

    if(__data_n > 0) {
        if((data = (void**)calloc(__data_n, sizeof(void*))) ==
            NULL) throw BadAllocException();
        if(__data != NULL) memcpy(data, __data, __data_n*sizeof(
            void*));
    }
    else data = NULL;
    if(__res_n > 0) {
        if((results = (void**)calloc(__res_n, sizeof(void*))) ==
             NULL) throw BadAllocException();
    }
    else results = NULL;

    #ifdef M2DF_DEBUG
    pthread_mutex_lock(&global::comm_mutex);
    std::cout<<"Task <"<<gid<<", "<<tid<<"> created successfully
        !"<<std::endl;
    std::cout.flush();
    pthread_mutex_unlock(&global::comm_mutex);
    #endif
}

_task::_task(const _task& __t) throw(BadAllocException)
    : gid(__t.gid), tid(__t.tid), uid(__t.uid), \
      code(__t.code), data_n(__t.data_n), res_n(__t.res_n),
         copies_cnt(__t.copies_cnt), _mutex(__t._mutex)
{
    pthread_mutex_lock(&_mutex);
    (*copies_cnt)++;
    pthread_mutex_unlock(&_mutex);
    data = __t.data;
    results = __t.results;
}

_task::~_task()
{
    pthread_mutex_lock(&_mutex);
    --(*copies_cnt);
    unsigned copies_n = *copies_cnt;
    pthread_mutex_unlock(&_mutex);

    if(copies_n == 0) {
        cout<<"DELETING TASK("<<tid<<"  "<<gid<<")"<<endl;
        delete copies_cnt;
        if(data != NULL)
```

```
            free(data);
        if(results != NULL)
            free(results);
        pthread_mutex_destroy(&_mutex);
    }
}

_task& _task::operator=(const _task& __t) throw(
    BadAllocException)
{
    if(this == &__t) return *this;

    gid = __t.gid;
    tid = __t.tid;
    uid = __t.uid;
    code = __t.code;
    data_n = __t.data_n;
    res_n = __t.res_n;
    copies_cnt = __t.copies_cnt;
    _mutex = __t._mutex;

    pthread_mutex_lock(&_mutex);
    (*copies_cnt)++;
    pthread_mutex_unlock(&_mutex);
    data = __t.data;
    results = __t.results;

    return *this;
}

} //end m2df namespace
```

## Utils

Next Listings contains some utility functions and macros mainly used for debugging concerns.

`m2df_utils.h`

```
#ifndef UTILS_H
#define UTILS_H 1
#include "m2df_debug.h"

#include <errno.h>
#include <cstdlib>
#include <cstring>
#include <pthread.h>
#include <queue>

#ifdef M2DF_PIPES_VERSION
#include <unistd.h>
#endif
```

```

#define Edge pair< unsigned, unsigned >

using namespace std;

namespace m2df
{
/**
 * \brief Utility function used to print pthread errors.
 * \param error_code Code of the error to be printed (errno value).
 */
void prerror(int error_code);

} //end m2df namespace

#endif //UTILS_H
```

m2df_utils.cpp

```
#include "m2df_utils.h"

using namespace std;

namespace m2df
{

void prerror(int error_code)
{
    switch (error_code) {
        case EAGAIN: cout << "EAGAIN";
        break;
        case EBADF: cout << "EBADF";
        break;
        case EBADMSG: cout << "EBADMSG";
        break;
        case EINTR: cout << "EINTR";
        break;
        case EINVAL: cout << "EINVAL";
        break;
        case EIO: cout << "EIO";
        break;
        case EISDIR: cout << "EISDIR";
        break;
        case EOVERFLOW: cout << "EOVERFLOW";
        break;
        case ENXIO: cout << "ENXIO";
        break;
        case ESPIPE: cout << "ESPIPE";
    }
}

} //end m2df namespace
```

# Bibliography

[1] Marco Vanneschi. *Architettura degli Elaboratori.* Edizioni PLUS, 2009.

[2] Jack B. Dennis Data Flow Supercomputers. In *IEEE Journal*, November 1980.

[3] Christopher Olston, Benjamin Reed Adam Silberstein and Utkarsh Srivastava. Automatic Optimization of Parallel Dataflow Programs

[4] Krishna M. Kavi, Hyong-Shik Kim, Joseph M. Arul and Ali R. Hurson. A Decoupled Scheduled Dataflow Multithreaded Architecture. In *Fourth International Symposium on Parallel Architectures, Algorithms, and Networks*, I-SPAN '99, pages 138–143, 1999.

[5] Krishna M. Kavi, Joseph M. Arul and Roberto Giorgi. Execution and Cache Performance of the Scheduled Dataflow Architecture. In *Journal of Universal Computer Science*, vol. 6, no. 10, pages 948–967, 2000.

[6] Krishna M. Kavi, Roberto Giorgi and Joseph M. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. In *IEEE TRANSACTIONS ON COMPUTER*, 50(8):834–846, 2001.

[7] Joseph M. Arul and Krishna M. Kavi. Scalability Of Scheduled Dataflow Architecture (SDF) With Register Contexts. In *Proceedings of the 2002 5th International Conference on Algorithms and Architectures for Parallel Processing*, pages 214–221, 2002.

[8] Roberto Giorgi, Zdravko Popovic and Nikola Puzovic. DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems.

[9] Costas Kyriacou, Paraskevas Evripidou and Pedro Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 17:1176–1188, October 2006.

[10] Kyriakos Stavrou, Demos Pavlou, Marios Nikolaides, Panayiotis Petrides, Paraskevas Evripidou and Pedro Trancoso. Programming Abstractions and Toolchain for Dataflow Multithreading Architectures.

[11] Pedro Trancoso, Costas Kyriacou and Paraskevas Evripidou. DDM-CPP: The Data-Driven Multithreading C Pre-Processor.

[12] Kyriakos Stavrou, Marios Nikolaides, Demos Pavlou, Samer Arandi, Paraskevas Evripidou and Pedro Trancoso. Tflux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 25–34, Washington, DC, USA, 2008. IEEE Computer Society.

[13] Costas Kyriacou, Paraskevas Evripidou and Pedro Trancoso CacheFlow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading In *Euro-Par 2004 Parallel Processing*, pages 561–570, 2004.

[14] Neungsoo Park, Bo Hong and Viktor K. Prasanna Analysis of Memory Hierarchy Performance of Block Data Layout In *Proceedings of the 2002 International Conference on Parallel Processing*, ICPP '02, pages 35–45, Washington, DC, USA, 2002. IEEE Computer Society

[15] Jakub Kurzak, Hatem Ltaief, Jack Dongarra and Rosa M. Badia. Scheduling linear algebra operations on multicore processors.

[16] Alfredo Buttari, Julien Langou, Jakub Kurzak, Jack Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. In *LAPACK working note #190.*

[17] Marco Vanneschi The programming model of ASSIST, an environment for parallel and distributed portable applications. In *Parallel Computing*, 28(12): 1709–1732, 2002.

[18] Marco Aldinucci, Marco Danelutto and Patrizio Dazzi. Muskel: an expandable skeleton environment. In *Scalable Computing: Practice and Experience*, 8(4):325–341, December 2007.

[19] Alessio Bonfietti, Luca Benini, Michele Lombardi and Michela Milano. An Efficient and Complete Approach for Throughput-maximal SDF Allocation and Scheduling on Multi-Core Platforms. In *Design Automation & Test in Europe*, DATE '10, pages 897–902, Dresden, Germany, 2010.

[20] Nan Guan, Martin Stigge, Wang Yi and Ge Yu. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 245–254, Grenoble, France, 2009. ACM.

[21] Shankar Ramaswamy and Prithviraj Banerjee. Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler. In *Proceedings of the 1993*

*International Conference on Parallel Processing - Volume 02*, ICPP '93, pages 134–138, Washington, DC, USA, 1993. IEEE Computer Society.

[22] Polaris Project Home Page.
*http://www.ecn.purdue.edu/ParaMount/Polaris/*

[23] OpenMP Home Page.
*http://openmp.org/wp/*

[24] OpenMP Wikipedia Page.
*http://en.wikipedia.org/wiki/OpenMP/*

[25] Cilk Project Home Page.
*http://supertech.csail.mit.edu/cilk/*

[26] libFLAME Home Page.
*http://z.cs.utexas.edu/wiki/flame.wiki/*

[27] ATLAS Project Home Page.
*http://math-atlas.sourceforge.net/*

[28] Intel. Intel® Xeon Processor Family.
*http://www.intel.com/en_uk/business/itcenter/products/xeon/index.htm*

[29] Intel. Intel® QuickPath Technology.
*http://www.intel.com/technology/quickpath/*