Master Thesis

Efficient multicore implementation of NAS benchmarks with FastFlow

Candidate
Dean De Leo

Supervisor
Prof. Marco Danelutto

Academic Year 2011/2012
Alla mia *famigghia*

... and to all of the people who have supported me
Chapter 1

Introduction

The thesis describes an efficient implementation of a subset of the NAS Parallel Benchmarks (NPB) for the multicore architecture with the FastFlow framework. The NPB is a specification of numeric benchmarks to compare different environments and implementations. FastFlow is a framework, targeted to shared memory systems, supporting the development of parallel algorithms based on the structured parallel programming. Starting from the NPB specification, the thesis selects a subset of the NPB algorithms and discerns an efficient implementation for both the sequential and parallel algorithms, through FastFlow. Finally, experiments on a couple of machines compare the derived code with the reference implementation, provided by the NPB authors.

The NPB is a public specification of algorithms to evaluate the performance of a specific architecture, framework and/or implementation. The algorithms are mostly of type numeric, inferred from common Computational Fluid Dynamics (CFD) applications. The specification encourages implementers to develop highly tuned codes, exploiting the details of the underlying platforms. The NPB authors provide a reference implementation in Fortran 77 and OpenMP as baseline to compare the results obtained with different implementations.
FastFlow is a programming framework embracing the concepts of structured parallel programming. Targeted to shared-memory systems, it allows to delineate parallel computations, in particular oriented to streams, through basic primitive patterns (skeletons). The pursued objectives are expressiveness and efficiency. The former means that a developer should be able to convey its computation through the native mechanisms supported by FastFlow. The latter entails the framework should experience a negligible performance penalty due to the offered abstraction.

The thesis describes a subset of the NPB kernels. It deeply analyses the computational aspects. It recognises the most critic points and proposes effective solutions to overcome the discerned issues. It takes into account how to tune up the algorithms for the C++ language. It devises a cost model for the parallel algorithms and outlines a practical and opportune mapping to the FastFlow framework. The final aim is to derive an efficient implementation of the selected benchmarks for the FastFlow scenario.

Conducted experiments will show several achievements, compared to the NPB reference implementation. First of all, for the examined cases, the derived C++ implementation will not experience any penalty respect to the Fortran 77 code, which is an issue pointed out by some previous works [14, 20]. Actually, in the sequential parts, experiments will prove enhancements due to algorithmic improvements and optimisation reasons. In the parallel scenario, FastFlow will reach good, if not near optimal, scalability values.

The thesis attempts to introduce the reader step by step to the fulfilled work. Chapter 2 gives more details about the context, providing a deeper overview of the NPB benchmarks and of the FastFlow framework. It also deals with the potential issues of the C++ language as opposed to Fortran 77, in the numeric computations. Chapter 3 presents the common adopted methodology, discussing the usage of caches, the regarded optimisations in C++, the cost model and its mapping in FastFlow. Chapter 4 describes the selected kernels, providing a computational analysis. Chapter 5 reports the experiments executed on a pair of multicore ma-
chines. Lastly, chapter 6 concludes the thesis.
Chapter 2

Context

This chapter aims to introduce the reader to the involved context. The thesis describes an efficient implementation of a subset of the \textit{NAS benchmarks} with the \textit{skeleton based programming} environment \textit{FastFlow}. Section 2.1 presents the main characteristics of the NAS benchmarks, while section 2.2 shows an overview of FastFlow and of its skeleton approach. At the end, we will experimentally compare the results of our implementation in C++/FastFlow versus the reference programs, provided by the NAS authors, in Fortran77/OpenMP. Section 2.3 highlights the entailments of these two diverse environments, in the examined context.

2.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) is a set of benchmarks suitable to evaluate the performance of a specific parallel architecture, framework and/or implementation [4]. These benchmarks rely mostly on numeric algorithms, representative of large scale of computational fluid dynamics (CFD) applications. The algorithms are specified with a \textit{pencil and paper} approach. An implementer should take advantage of the details of its underlying platform to provide an optimised
implementation for its particular environment. Among all types of benchmarks, the thesis concerns a subset of the NPB kernels. The paper [5] is the reference specification for the examined problems.

At the beginning of ’90, the NAS research group proposed a set of benchmarks to evaluate the parallel systems. The Numerical Aerodynamic Simulation (NAS) program, based at the NASA Ames Research Centre, had the necessity to invest in the acquisition of new machines. At that time, new heterogeneous distributed systems were arising, with their peculiar characteristics and advantages. The NAS group judged the previous existing benchmarks, targeted to supercomputers, unsuitable to the emerging context. To compare the different architectures, they designed a new set of problems, derived from their application area, CFD, to properly measure the performance.

Vendors, such as IBM, Cray, Intel and others, implemented the benchmarks in their architectures and submitted the results [28]. Taking advantage of the peculiarities of their systems, the vendors adapted the algorithms to their environment to fully exploit their specific environment. As the systems became more and more powerful and new scenarios arose, the NAS evolved the set of the NPB problems. They included new sizes of problems and created new types of benchmarks to stress particular areas of a system.

Besides the machine vendors, research groups tested new frameworks and alternative implementations on specific architectures. To this purpose, the NAS group provides a reference implementation in Fortran 77 for both shared-memory and distributed systems. The shared memory implementation employs OpenMP as multi-threading framework [22]. The NAS group regularly updated and improved their implementation in the years, releasing a portable but rather optimised code.

The benchmarks are specified only in terms of the algorithmic method to implement. The specification describes the input parameters, the algorithm and the constraints to respect. The algorithms are deterministic. To ensure the correctness of the implemented methods, the algorithms yield a small output that must
agree with given reference values (verification test). The specification explicitly delineates how the timing must be performed, when it must start and end.

The specification lists a (reasonable) set of global constraints that every implementation must conform. The most important rules, meaningful to the thesis context, are:

- No low level (assembly) code is allowed.
- Numeric calculus must be carried on 64-bit double precision numbers.
- External libraries are allowed only if written in a high level language (Fortran, C, C++).

Refer to [5, 6] for the whole list of restrictions.

The NPB kernels solve a single and circumscribed problem. Their aim is to test specific features of the platform. The kernels are more limited algorithms compared to the NPB simulations. The simulations represent a whole application. They require a deep knowledge of their related domain and more a substantial effort to be implemented. The thesis takes into account three representative kernels. Each selected kernel exhibits its own peculiar characteristics, that significantly distinguishes it from the others.

The thesis focuses on the class C kernels for the shared-memory environment. The NPB specification sets several classes for the same kernel, through the definition of different input parameters. The main entailment is the size of the problem. The class C kernels are the more appropriate for the involved scenario. For the examined machines, the serial kernels required 1 ∼ 6 minutes to complete the algorithm and up to several gigabytes of main memory, depending on the specific problem.

A fair literature exists on the topic of the NPB kernels. Vendors were rather reluctant in the publication of their implementations, preferring to keep their results
confidential [6]. The paper by Agarwal et al. [1] discerns a valuable analysis of the NPB kernels. The work by Ferrari et al. [18] may represent another interesting source. Among the others, some NAS reports [6, 29] provide a generic analysis of the kernels. However, they are largely targeted to distributed systems.

In the shared-memory scenario, the main reference is the OpenMP implementation provided by the NAS authors [22]. Frumkin et al. [20] ported the OpenMP reference in Java. They performed a raw translation of the Fortran 77 code, achieving poor results. Notable, the UPC group [10, 14] has put several efforts in the implementation of the NPB kernels in C/UPC. Though optimised, the sequential part of their implementation in C is still a porting of the Fortran 77 NAS code, which entailed an intrinsic as unfavourable delay in their comparisons.

The approach of this thesis is rather different w.r.t. mostly previous works. It does not involve a translation of the Fortran 77 codes provided by the NAS authors. Indeed, it fully reviews the selected kernels starting from the NPB specification. It presents a formal analysis of the kernels, recognising the critical points, and proposes solutions to overcome the discerned issues. The implemented algorithms are effectively optimised for the multi-core environment taken into account. The final experimental comparison within the OpenMP reference codes will show several achievements.

2.2 FastFlow

FastFlow is a programming framework suitable to support the implementation of parallel algorithms onto shared memory multi-cores. It enables the programmer to model its computation by structured design patterns (skeletons) based on streams. Developed by the Universities of Pisa and Turin, it is available as a C++ library. The reader may refer to [2] for a broader overview, while [3] presents a guide on its usage. The thesis implements the parallel algorithms on top of FastFlow.
The framework proposes three different layers. The lowest level is characterised by the explicit access of both the synchronisation mechanisms and the queues that the framework implements. The intermediate layer permits to represent arbitrary data-flow graphs, composed by logical stages. The communication may take place through the explicit placing of data channels. The thesis makes use of the third and higher abstraction layer, where a developer contrives its computation in a limited set of common skeletons. In [32], this methodology is referred as structured parallel programming.

FastFlow provides two primitive skeletons: the pipeline and the farm. The pipeline, depicted in Figure 2.1a, expresses the connection of single stages in a cascading sequence. The stages may be sequential modules or other parallel systems. The communication among the stages is feasible through explicit data channels. In FastFlow, a farm allows to replicate and/or split the computation of a single stage among a set of multiple workers.

In general, three different types of entities conceptually compose a farm [32], see Figure 2.1b. The Emitter module dispatches the tasks to compute to available workers, according to a certain scheduling policy. The Workers perform the actual processing of received tasks. Finally, the Collector gathers the computed tasks from the Workers and, if required, forwards them to the next stage of the system. Both the emitter and the collector are service nodes, that may be merged into a single module. In this case, the terminology Master / Workers (or Master / Slaves) applies.

In FastFlow, a developer can customise the behaviour of the emitter, the collector and the set of the workers. Actually, the definition of the collector is not mandatory. When not specified, the workers act as sink, and must consolidate in some way their results in memory. In this situation, the pattern may resemble the mentioned Master/Workers paradigm. All the other paradigms, such as MapReduce, Divide&Conquer, ForAll, may be obtained through the composition and/or nesting of the primitive FastFlow skeletons.
Figure 2.1: Primitive skeletons in FastFlow. The figure (a) depicts a three stages pipeline while (b) is a farm composed by the Emitter $E$, a set of Workers $\{W_1, ..., W_n\}$ and the Collector $C$ modules.

FastFlow has been designed with the efficiency goal in mind. It attempts to reduce the unavoidable and additional overheads that parallel algorithms entail, by minimising the need of synchronisation locks and barriers. The framework is aware of the underlying architecture and typology (e.g. NUMA or SMP), featuring peculiar implementations for particular cases and scenarios. Finally, it gives the opportunity to further tune the program for the environment, such as a uniform mechanisms to **pin** the resources to physical nodes.

The thesis relies on an approach based on the structured parallel paradigms. They are ultimately implemented by taking advantage of FastFlow. However, the framework has been conceived to operate on streams of multiple elements, while the selected benchmarks, except the kernel EP, are more suitable to process a single item. Our solution was to map the logical computational graph in a peculiar Master/Slaves paradigm. The section 3.3.2 copes with this issue. At the end, in the experimental results, the derived implementations will exhibit a performance increase compared to the reference OpenMP programs, provided by the NAS authors.
2.3 C++ vs Fortran 77

The ultimate result will be the comparison of the derived implementations with the reference NPB codes [22]. The two background environments are very diverse. The NAS developers judged the Fortran 77 (F77) language more suitable to represent their numeric computations, employing OpenMP for the shared-memory context. Instead, the implementation derived in this thesis is in C++, with the usage of FastFlow framework. Dealing with these different premises would deserve a closer examination of their implications.

Fortran 77 is actually a language aimed to numeric calculus. For what concerns a comparison in performance with C++, limited to the examined benchmarks, the key point is the static memory model that F77 adopts. A F77 compiler has potentially full knowledge of the memory that the program may touch. It might always track to what a pointer refers. For this reason, it can release a more optimised code for the delineated computation. It can do more, properly aligning and displacing the memory in a more efficient manner. The major drawback of a static memory model is it severely limits the expressiveness. The F77 syntax is somewhat old fashioned, too.

Similarly, a compiler natively supports OpenMP. Indeed, in OpenMP, the developer indicates how to parallelise a loop construct through the usage of macro directives. The compiler recognises the directives and emits a proper code. Taking advantage of the knowledge of the related environment, it can release a fine tuned and specific code. It may also avoid to parallelise a certain part of code, if reckoned unfavourable.

C++ was not designed as a language targeted to numeric computations. It is more eligible to a dynamic memory model. By default, this choice inhibits several optimisations that a compiler would execute. The related assembly code could be more polluted, to take into account different cases that may come up. In the years, a popular trick was to go back to the static memory model, exploiting techniques
such as template meta-programming [7, 19]. A C++ compiler could deduce and perform the same optimisations managed by a F77 compiler. The price to pay is again a reduction of the achievable expressiveness.

A C++ compiler is agnostic regarding the behaviour of FastFlow. The choice of how to introduce the parallel tasks and of what *grain* they represent is completely left to the developer. If he or she wrongly predicts the weight of its computation, then a deterioration of the performance will likely occur. Making parallel an unsuitable task, as its grain is too fine, will worsen its execution time. Nevertheless, compared to OpenMP, FastFlow allows additional constructs to explicitly tune the computation for a specific scenario.

The thesis considers the algorithms in a dynamic memory context. For the examined benchmarks, it was always possible to plug the gap that Fortran apparently shows. Some non obvious problems, notably the aliasing and the vectorisation, have to be treated. A cost model is needed to understand where and how it is favourable to parallelise. Moreover, the algorithmic analysis brought in newer aspects to improve. At the end, the extra flexibility that C++ and FastFlow expose, will pay for.
Chapter 3

Methodology

A common methodology has been adopted in the study of the examined benchmarks. Initially, starting from the NPB specification [5], a sequential code in C++ has been directly generated. The result was usually an implementation 1.5x ∼ 3x slower than the reference Fortran77 code, provided by the NPB authors. Nevertheless, this step served two goals. The former was to gain a better understanding of the problem domain. The latter, to detect additional issues that might be hidden in the first analysis of the problem specification. From this knowledge, an algorithmic design was properly deduced.

There are several aspects to take into account. The NPB specification delineates the algorithmic scheme to apply, and the constraints to respect. The free choices concern the way the computation is arranged, the employed data structures, the actual implementation of involved operators and the whole description of the parallel algorithm. Both memory and caches usage are key factors, covered in the thesis, that strongly contribute in the overall performance of the algorithm.

Section 3.1 describes the impact of memory accesses on the execution time. It introduces the I/O model to analyse the behaviour of the algorithms related to their potential cache faults. The section presents a simple experiment to illustrate
the cost of the sequential versus the random scansion of the memory elements. The analysis of the examined benchmarks will occasionally refer this section to justify some design choices.

C++ is not a language specifically designed to execute scientific calculus. Some optimisations are required to compete with an equivalent Fortran 77 implementation. Section 3.2 describes two essential techniques: restricted pointers and automatic vectorisation. Taking advantage of these two methods, for the examined benchmarks, the C++ implementation could at least compete on the same level with the equivalent Fortran 77 code.

The parallel algorithms derive from their sequential counterparts. A cost model, based on the structured parallel paradigms [32], has been built. The model aimed to structure the parallel algorithm and to predict its performance. Eventually, the model will be implemented in FastFlow through a peculiar Master/Slave paradigm. Section 3.3 addresses these issues.

### 3.1 Minimising the memory faults

The accesses to the main memory play a major role in the final count of the completion time. While all proposed NPB algorithms have a linear complexity, related to the amount of involved items, they can be readjusted to decrease the penalty due to random accesses in the memory. To the purpose, the I/O model can serve as theoretical abstraction to analyse this task. Eventually, the algorithms should be tailored to work for an optimal “working set”, possibly acquiring the required data through sequential scansion.

The I/O model (or external memory model [17]) provides an approximation on the number of “faults” sustained by an algorithm. The model recognises a hierarchy of two memory levels. The lower level is characterised by a limited capacity while the upper by an unbounded size. The algorithm acquires the data
in contiguous blocks, which must reside in the lower memory level during their elaboration. The final cost is basically given by the count of the accessed blocks (i.e. faults) during the lifetime of the algorithm. The objective is to eventually ensure both spatial and temporal locality.

In this context, the hierarchy is represented by one of the cache levels and the main memory. The disk accesses are not examined, as for the class C of evaluated kernels the affected data will always fit in the main memory. The working set has to be adjusted to reflect both the cache capacity and the required blocks predicted on the model. Nevertheless, the I/O model does not take into account if the algorithm accesses the requested data sequentially or randomly. The two approaches have different costs and, if possible, the former should be preferred.

To figure out the role played by the distribution of memory accesses, a simple experiment has been conducted. It follows the same tests performed by Drepper [12], indefinitely iterating over an array of elements in both sequential and random order. The array is composed of pointers referencing to the same elements of the array (see Figure 3.1). In the sequential test, each pointer references to the next element, but the last item that points to the first position. In the random access test, the pointers are randomly permuted in the positions of the array, ensuring that each element is referenced by one pointer.

The test has been conducted on the machine “Andromeda” of the Computer Science department. Andromeda is a shared-memory NUMA architecture characterised by two interconnected homogeneous nodes with their own local memory modules (see chap. 5, figure 5.1a). Each node features a cache hierarchy of three
levels, where the first level has a capacity of $2^{15}$ bytes (only data), the second level $2^{19}$ bytes and the last a capacity of $2^{23}$ bytes. Obviously, distinct machines could provide diverse outcomes. The test aims to give a generic overview on the difference of the access times.

Figure 3.2 shows the results for this experiment. The abscissa represents the working set, given by the size of the array times the size of the elements/pointers, that is 8 bytes on this machine. The ordinate shows the average access time on each iteration per element. There are several conclusions that can be inferred from the graph:

1. As far as the working set totally resides in the first cache level, there is no difference if the access is sequential or random. This is the most favourable case.

2. Scanning data sequentially is fast no matter what the working set size is or where the data is located. Indeed the lines related to the local and remote memories overlap almost everywhere. The average time per iteration is around $1.8 - 2.5$ nsecs.

3. Depending on the size of the working set, the access time for the random scenario is delimited by the latency of the cache that can hold all the required data. If the cache hierarchy cannot contain the whole working set, then the access time indefinitely increases.

4. If the working set is smaller than the cache capacity, then it does not matter if the data is allocated locally or remotely, as the processor will directly acquire it from the local cache. However, for greater working sets, compared to the local memory, randomly retrieving the data remotely entail a penalty of $\sim 30 - 40\%$.

The sequential and random tests represent the optimal and worst case possible scenarios, respectively. Actually, retrieving the data “linearly”, but without processing (touching) all items into the fetched block, entails higher access times [12].
Figure 3.2: Results of the test on sequential and random accesses for the machine “andromeda” of the Computer Science department. The continuous lines are the access times relative to the local main memory, while the dashed lines to the remote memory. Note that the lines for the sequential accesses overlap almost everywhere. The grey vertical lines are convenience references to indicate the capacity of the three cache levels.

Nevertheless, scanning the data sequentially, or linearly, cannot be always ensured, due to intrinsic structure of the dealt problems. When forced “to jump” on some separate zone of the memory, it is profitable to utilise all the data fetched, trying to avoid to newly acquire it once the related block has been removed from the cache. In this manner, the latency to transfer a block can be amortised and no acquired data remain unused.

Some additional concerns have to be taken into account for the parallel algorithms. Indeed, there are more potential factors that could degrade the performance. As pointed out above, if the local cache cannot contain all the remotely allocated data, the option to explicitly transfer it on the local node should be con-
sidered. Moreover different processors may share some parts of the cache hierarchy, decreasing the effective dimension of the optimal working set.

Summing up, the memory layout and the acquisition of data have a fundamental impact on the final performance of the algorithms. With the aim of minimising the expected faults, an abstraction with the I/O model can be analysed. Likely, it will be convenient that algorithms proceed by blocks. Once the blocks and the partitions of data composing the working set have been determined, they should be set to fit in the maximum cache capacity of the underlying architecture. Whether possible, sequential scansions of the involved data should be favoured.

3.2 Optimisation

Optimisation is the process of tuning the implementation to a target architecture. C++ is a natural language to apply optimisations as a programmer can keep a strict control of the environment, where necessary. The usage of custom objects is unavoidable in practice, to maintain the opportunity to apply significant implementation dependent optimisations, and, at the same time, to provide a handy interface to external users. Section 3.2.1 discusses this topic.

The developed implementation does not employ perverse tricks. Most of the optimisations are more good sense practices, such as avoiding redundant expressions, minimising the type casts, inlining critical parts of code, avoiding the price of abstraction where excessive. Vice versa, the implementation does not leverage on more controversial techniques, such as explicit loop unrolling, abrupt jumps (goto) in the code, explicit branch prediction and manual cache prefetching. The rationale is to expect the compiler would perform, if convenient, these lower level patterns.

Restricted pointers and alignment of the dynamic memory are two essential and non obvious optimisation techniques. Sections 3.2.2 and 3.2.3 cover these aspects.
They are not general techniques, but targeted to the specific environment for the performed experiments. In particular, the platform is assumed to be Linux (or Posix compatible), over an x86_64 architecture, and the compiler has to be GCC 4.7 or higher.

3.2.1 Representation of the mathematical structures

This section discusses how to represent the involved mathematical vectors, matrices and grids that kernels require. Raw C/C++ arrays can effectively express unidimensional vectors. Linear operations are simple enough to convey that can be straightly written. However, flat usage of raw C arrays to model more complex matrices could result inadequate. Similarly, the standard library `std::valarray` data structure does not seem to satisfy both performance and flexibility requirements. It is more convenient to build custom data structures that, depending on problem domain, can afford a highly optimised implementation.

Direct usage of raw C arrays might result rigid, ineffective and cumbersome. The classic C matrices demand the knowledge of all but first dimensions at compile time [31]. Thus, they do not fit in a dynamic memory context, as established in the thesis objectives. To overcome this issue, an alternative approach is to dynamically allocate the matrix through array of pointers [24]. The idea is define a array of pointers for each dimension d, that refer to the pointers of the next dimension, while the last dimension actually contains the values. This technique is rather ineffective, due to the potential and uncontrolled jumps in the memory that it may introduce.

The C++ standard library provides the `std::valarray` data type to perform numeric calculus [31]. The interface enables the common arithmetic expressions as additions, multiplications, assignments, etc. An important feature is the possibility to easily represent matrices of arbitrary dimension, simple partitions or a sequence of values with holes, through the usage of the `slices`. A compiler should offer a
strongly optimised implementation of this data structure, exploiting as much as possible the vectorisation of the arithmetic expressions and/or other sharp tricks that the underlying platform may support.

Conducted experiments based on valarrays were unsatisfactory. It does not seem that the default implementation coming with the G++ compiler is strongly optimised, at least for the environment taken into account. The same computations performed on raw C arrays turned out to be always faster. Moreover, the data structure does not offer a flexible enough interface to support efficient retrieving of the values in a custom manner, as required by the developed algorithmic models. The order of the memory accesses is fundamental to allow linear scansions of the values. The valarray seems to be very limited when coping with this issue.

The adopted solution is to build a custom and specialised data type for each mathematical structure. The data type should perform a single allocation for the whole structure (or partition) of values reckoning with its global dimension. For instance, a 3D grid of dimension $N^3$ entails an array of size $N^3$. The value at row $i$, column $j$ and depth $k$ will be mapped at position $k \cdot N^2 + i \cdot N + j$. Moreover, in this way it is possible to strictly control the padding among the values, to potentially enhance the vectorisation instructions (see section 3.2.3).

The custom data type should provide a method for each of the possible operations. It should take advantage of the memory representation of the data structure, scanning values once in their native form. On the one hand, having a knowledge of the raw data structure, operations can be manually implemented in a highly optimised flavour. On the other hand, the same data structure can provide a handy interface to external users.

### 3.2.2 Aliasing

In C++, aliasing refers to the circumstance that two or more different pointers may refer to the same memory location [15]. From the point of view of a compiler, this
eventuality may disable a full set of optimisations that could be actually fulfilled. Compilers perform a static analysis of the pointers inside the code, attempting to detect active aliases. When they are unable to figure out if aliases exist, they must conservatively assume aliasing is present. For this reason, it is opportune to help the compiler explicitly expressing that, in certain cases, pointers do not actually alias each other.

Assuming aliasing is present, when it is not, is a major source of performance deterioration. Actually, it causes the emission of polluted code, that reloads values from the memory when they are already in a register, or stores back values in the memory when holding them in a register would have sufficed. Moreover, assuming the two pointers are aliases, also entails that all derived pointers might be aliases, creating a domino effect. The scenario that compilers are trying to handle, comes from the possibility of altering the values referred by a pointer and eventually accessing them through an alias.

For instance, consider the following snippet of code, taken from [23], representing the assignment to the vector $A$ of the constant value $1 - c$:

```
Listing 3.1: Aliasing example

void assign(double A[], const double *c) {
    for (int i = 0; i < N; i++) {
        A[i] = 1 - (*c);
    }
}
```

A desirable optimisation would be to hoist the value $1 - (\star c)$ in a temporary $t$ and perform the assignment $A[i] = t$. However, from a syntactic point of view, in C++ it is legal to invoke the function `assign(A, &A[10])`. Clearly, a such perverse eventuality would break the above optimisation; and a compiler must cope with this case. Note how the `const` specifier does not solve the issue. It just expresses that the memory location cannot be altered through the pointer $c$, not
that the memory pointed by \( c \) cannot change at all.

In contrast, aliasing cannot occur in Fortran 77. By default, no two identifiers in a given subroutine may refer to the same storage location, unless explicitly stated by the keyword EQUVALENCE [23]. This behaviour allows Fortran compilers to perform more aggressive optimisations than most modern C/C++ compilers. It is one of the major reasons why Fortran code might be more efficient than an analogous computation expressed in C++. To be able to reach the same performances, it is necessary to address this issue.

Simpler cases such as the listing 3.1 can be treated by manually hoisting the possible source of aliases into local temporaries. This solution is not feasible for more complex and practical situations, such as a matrix multiplication \( A = B \times C \). Some compilers, but currently not GCC [19], offer an option to disable the aliasing in the whole program. This is likely a burden overkill. The programmer must ensure that aliasing can never occur, even in areas where the improved performance is irrelevant or negligible. Sometimes the aliasing could result in a worthwhile feature, permitting, for instance, to compute \( A = B \times B \) with the same function.

Restricted pointers allow to specify circumstantial scopes where aliasing cannot take place. They are part of the C ANSI/ISO 1999 specification, but not strictly in standard C++. However, the feature is so important that almost all common C++ compilers (e.g. GCC, Intel ICC, Microsoft, Kai, etc) support it as a language extension. In C++, a restricted pointer is declared through the _restrict type specifier, after the related pointer. Considering the function declaration of listing 3.1, it becomes
\[
\text{void assign(double* _restrict A, const double *c)},
\]
meaning that, in the current scope, the only way to access values referred by \( A \) can be exclusively accomplished through the pointer of \( A \).

A restricted pointer is a contract between a programmer and the compiler. The programmer must guarantee that the only way to access, in the current scope, the referred object is exclusively through the restricted pointer \(*p\) or any other derivated pointer originating from \(*p\). It is permitted to use multiple restricted
pointers to access different parts of the same object or array. The restriction demands that the involved partitions do not overlap each other. A restricted pointer inhibits the existence of any potential alias, originated from an external scope, and allows compilers to perform more aggressive optimisations.

Restricted pointers are the first ingredient to reach the same execution times achievable by Fortran 77 programs. They are basically what the `std::valarray` should use under the hood. The second ingredient is the alignment of the dynamic memory, covered in the next section. Once these two topics are managed, a C++ numeric program may compete on the same efficiency level, without sacrificing much of the expressiveness that a Fortran 77 program does not offer.

### 3.2.3 Automatic vectorisation

This section concerns with the application of SIMD instructions in a numeric scenario [19]. Nowadays, modern processors, based on the architecture x86_64, are able to perform arithmetic operations on multiple elements within a single instruction. This form of parallelism is named Single Instruction Multiple Data (SIMD). A programmer has to satisfy some memory requirements to fully exploit this feature. In the context of the examined benchmarks, taking advantage of the vectorisation of arithmetic expressions will be fundamental in the kernel MG.

The number of elements that may be simultaneously processed depends on the primitive data type and on the available instruction set. Almost all x86_64 processors should have access to the Streaming SIMD Extensions 2 (SSE2) instruction set. When the data types are doubles (64 bit), using the SSE2 instruction set, processors can perform up to 2 arithmetic operations within a single instruction. More recent cpus support the Advanced Vector Extensions (AVX) instruction set. With the double data type, the AVX permits to perform up to 4 operations within a single instruction.

There are two approaches to deal with this feature: automatic vectorisation
and explicit vectorisation. The former scenario arises when the compiler automatically emits vector instructions for the involved code. In the latter case, the programmer directly writes the code using the MMX or AVX wrappers, available through intrinsic functions. While explicit vectorisation leads to a stricter control of the produced code (specific to a certain instruction set), it is a very low level, error prone and cumbersome task. Besides that, it does not comply with the NPB rules. The only viable way remains automatic vectorisation.

To effectively exploit vectorisation, the memory must be aligned. SSE2 instructions require that given operands are aligned to 16 boundaries, i.e. their memory address is divisible by 16. Similarly, AVX instructions require operands aligned to 32 boundaries. When the memory is dynamically allocated, it is responsibility of the programmer to align the created object to satisfy the mentioned constraints. In Posix compatible systems, the programmer may accomplish this task through the function: `int posix_memalign(void **memptr, size_t alignment, size_t size)` [27]. To be more robust, in the developed code for the kernels, the memory is aligned to the line size of the cache of first level, that is typically 64 bytes.

The programmer should also inform the compiler when the memory is actually aligned. In some sense, this situation is similar to the issue of the aliasing: a compiler could not know to what memory area an external pointer refers. It assumes the conservative position that memory is not aligned. This causes the emission of polluted code that performs extra unnecessary checks, strongly degrading the performance. GCC 4.7 introduces the compiler specific keyword `_builtin_assume_aligned(const void *exp, size_t align)` [21], authorising the compiler to assume that memory is already aligned. As for the restricted pointers, it generates a contract between the programmer and the compiler. It is a duty of the programmer to guarantee that the specified address will be really aligned.

Both restricted pointers and automatic vectorisation are the key methods to
optimise the numeric computation. They need an additional attention by the side of the programmer. Once he or she is aware of their implications, the steps to follow are basically two: allocate the memory with `posix_memalign` and declare pointers both restricted and aligned. The compiler will do the rest when the flag `-O3` is enabled. The advantage in terms of speed up turns out to be impressive.

### 3.3 Parallel algorithm

The sequential algorithms are the basic blocks for the developed parallel counterparts. From the study of their algorithmic complexity, the timing of their computational steps and the identification of their interdependencies, the parallel algorithm may be derived as an adaptation of the sequential method. Section 3.3.1 outlines the generic scheme to achieve the parallel algorithm and to predict its performance in the experiments. Section 3.3.2 describes how to pass from the built model to the actual implementation, based on the FastFlow framework.

#### 3.3.1 Model

Cost models may be a favourable tool to analyse a parallel system. The kernel EP is simple enough that a model is not stringently required. It can be straightforwardly represented by a farm paradigm. For the other kernels, the study of a model is worthwhile. The objectives are essentially two. The former is to constitute a ground layer to reason and eventually develop the final implementation. The latter is to predict the performance in the experiments.

The developed cost models rely on the general macro data flow graph $G(N, E)$. The nodes $N$ represent the logical tasks that the computation has to perform. The set of tasks are always derived starting from the sequential algorithm. The edges $E$ are directed. They express the inter dependences among the nodes. Formally, the Bernstein conditions are an effective tool to safely assert when the dependences
hold [32]. Typically, for the examined benchmarks, tasks will form a strict ordering sequence. The resulting graph will be a chain of nodes, with exclusively one input and one output edge.

Our interest will concern the prediction of the completion time for the whole simulation. The computation involves a single element to process. There is no stream of multiple items to elaborate. For the examined benchmarks, the algorithm must always repeat a predetermined amount of iterations $NITER$. For instance, a benchmark may require the application of $NITER$ times of the conjugate gradient, or $NITER$ iterations of the multigrid method. Therefore, the overall completion time will be $NITER \cdot T_G$, where $T_G$ is the completion time of the data flow graph restricted to complete a single iteration. In the derived graphs, the parallel algorithm may advance by one stage at time. Therefore the overall completion time is given by:

$$T_C = NITER \cdot \left( \sum_{\forall n \in N} T_{compl}(n) \right)$$

(3.1)

We will not render all modules parallel. That will depend on the grain of the task. When the grain is too fine, say the system can already accomplish the task in 1 $\sim$ 10 milliseconds, then the module will remain sequential. To ease the calculus, the completion time of sequential modules can be summed up together. In the following, this quantity is referred as $T_{sequential}$. It is a constant, independent of the actual parallelism degree of the system.

When profitable, the modules may exploit a data-parallel paradigm. The workers, composing the module, will be responsible for a single partition of the whole related data structure. A worker owns its partition in the sense it is the only one entity that can alter its partition. In [32], this concept is referred as “owner computes rule”. Depending on the type of the computation, local or not, the paradigm may be a map or a stencil. The latter case entails communications among the workers to carry forward their computation. The service nodes emitter
and collector may be present to perform the standard collective communications, such as multicast, scatter and/or gather.

The communication among the nodes may introduce an additional overhead. In a shared-memory context, the communication may be accomplished passing the pointers to the original locations of data. In this case, we assume the communication cost is negligible compared to the execution time. Sometimes, it could be more convenient to physically move the actual data. This situation usually arises when the workers are located in different nodes of a NUMA system. For this scenario, we account the time to effectively copy a generic chunk of memory of the same size of the transferred data. In this context, the overlapping between calculus and communication cannot take place.

The completion time of an isolated parallel module is function of the parallelism degree \( p \). Ideally, the completion time would be \( T_{\text{seq}}/p \), where \( T_{\text{seq}} \) is the sequential time of the same computation. When the communication costs cannot be assumed negligible, they have to be added to the ideal completion time. The collective communications are modelled as sequential point to point communications. At the end, the formula 3.1 turns out to be parametric in \( p \).

### 3.3.2 Implementation

This section deals with the transformation from the model, described in the previous section, to the implementation, based on FastFlow. There are basically three reasonable approaches. The first is to directly map the graph of the previous section in FastFlow, inserting a feedback channel to link the last and the first module. The second is similar to the first, but it does not include the feedback channel. The third describes the mapping to an equivalent master / slave paradigm. This section explains the drawbacks of the first two approaches. At the end, a peculiar master / slave paradigm will be common to all developed kernels.

FastFlow is a platform aimed to depict structured computations based on
streams. It provides two basic skeletons: pipeline and farm. In the FastFlow perspective, a data parallel paradigm must be finally mapped into a farm [3]. A custom emitter may be provided, with the objective to scatter the data to available computational units. The scheduler should be explicitly specified, to keep a coherence between the emitter, that actually sends the data, and the scheduler itself, that selects the workers to who the data will be eventually sent. Similarly, a collector module may be provided, to gather the results from the computational units. The workers can be customisable too, keeping their own state. The pipeline paradigm allows to connect the different stages (nodes) of the parallel system (graph).

A possible natural mapping would be to directly represent the modelled graph in FastFlow. Each node in the graph would become a node in FastFlow. Single nodes can be transformed in intrinsic parallel modules exploiting the farm skeleton. As mentioned, data-parallel paradigms should be implemented by the farm. The pipeline would link the sequence of the stages, as described in the model. Finally a feedback channel would link the last stage with the first. This would create a cyclic graph, to model the amount of iterations $N_{ITER}$ that the algorithm should perform.

While this approach would be suitable for an actual data flow graph, it would be not applicable in this context. Due to the efficient, fine grain implementation of FastFlow, the nodes would remain in active wait, once activated. For a given parallelism degree, the system must ensure to have sufficient resources to accommodate the specified number of nodes. That would bring in a potential waste of resources. As this is a single data computation, only one stage of the system can perform some useful work at a given time. All the other nodes would remain in active wait, waiting for items in their queues. This situation would reduce the overall potential maximum parallelism degree the system could reach.

Figure 3.3 attempts to exemplify a simple graph with two stages. The first stage is a sequential module. The latter represents a map paradigm, implemented
as a FastFlow farm. It features both an emitter and a collector to scatter and gather the partitions, respectively. A feedback channel links the collector back to the sequential module. Given $N$ physical nodes, the maximum achievable parallelism degree by this system is $N - 3$. For instance, in a multicore machine with 8 processors, the maximum reachable parallelism degree is 5. Compared to an OpenMP implementation, the same map paradigm may be executed with up to 62.5% of the resources. The issue here is that the sequential module, the emitter and the collector must remain active also when they cannot process any item. Moreover, if the system encompassed more stages, the situation would become even more detrimental.

An alternative approach would be to remove the feedback channel. This scenario would produce a kind of wave effect. FastFlow would make active the stages once the item reaches their input channel. Since there is only one item to process, a stage would go away once it sends the computed element to the next stage, releasing the underlying thread. To perform the amount of iterations required by the benchmarks, the pipeline should be externally invoked $N_{ITER}$ times. Only one
stage of the whole pipeline might be active in a given time. This approach may represent an arbitrarily long graph, exploiting the maximum parallelism degree for each stage.

The major drawback of this solution is the overhead due to pinning. In the scenario depicted in figure 3.3, where a sequential phase alternates with a map paradigm, it would be a convenient property if the workers, when inactive, might keep their respective caches hot, with the data of the previous computation. Additionally, workers might keep a state, stored in the local memory of a certain NUMA node. It would be advantageous if the same workers would reside in the same cpu. This feature can be accomplished through the pinning, i.e. the explicit mapping of a worker to a specified cpu.

The pinning is a relatively costly procedure. In the case with the feedback channel, all stages would remain alive for all the lifetime of the computation. The pinning could be performed at initialisation time, not impacting the completion time. In the latter case, without the feedback channel, FastFlow would resume the required units every time needed. The pinning should be executed for each stage and for each iteration. That is rather inefficient. Finally, in a fine grain computation, the continuous dispatching of logical threads might negatively affect the performance.

The solution adopted in this thesis leverages on a generic master / slave paradigm. The master advances the current state in the modelled graph, scheduling the required computation to workers. In case, it performs the collective communications to available workers. The workers assume different roles, depending on the task assigned by the master. A single worker, say the first, is promoted to also execute the sequential stages. Observing that there is only one task in the whole pipeline of the previous section, then this implementation conforms to the described cost model.

There are several good points of this approach. The first is that there is only one service node, that is the master. The maximum achievable parallelism degree
of the system is $N - 1$. The second point is, as the workers remain always active, their caches are “hot”. Therefore they can immediately restore the computation decreasing the initialisation times. Finally, the pinning can be executed once, at the beginning of the computation.
Chapter 4

NPB Kernels

Three kernels were taken into account. Each problem presents different characteristics and peculiarities. Kernel EP regards the generation of pseudo-random deviates according to the Gaussian distribution. It is a coarse grain computation, with no inter-iteration dependencies, and the possibility to reach near optimal scalability. Kernels CG and MG concern the resolution of an algebraic linear system $Ax = b$ by two diverse methods. They involve operations among matrices and grids, with strong dependencies in the computation order. They are suitable to a data-parallel approach.

The description of each kernel is divided in three sections. The first is the specification of the algorithm to implement and the constraints to respect, according to the NPB reference [5]. The second is a careful analysis of the algorithm, covering the main properties, the general complexity, the memory access (see section 3.1) and the recognised critical points. It describes the adopted solutions to solve the discerned issues and analyses the characteristics of the parallel algorithm. Lastly, the third section discusses several implementation details that can relevantly affect the performance.
4.1 Kernel EP

The Embarrassingly Parallel (EP) kernel mainly aims to measure the peak performance of a single node. The required computation can be easily parallelised as it involves a fixed amount of iterations, predetermined in advance. The iterations are independent each other. Furthermore the parallel algorithm requires a scarce communication among the nodes, practically negligible in the total account of the completion time. The main focus for this kernel is the realisation of an efficient implementation of each iteration.

The required task concerns the generation of (pseudo) random deviates related to the standard Gaussian distribution. According to NAS authors, this problem is representative of a large class of Monte Carlo simulations. The method to generate the outcomes is fixed and fully described. The result of the computation is the sum of the created outcomes, along with the output of the maximum outcomes that lie into the interval between two consecutive integers. Together with the algorithm, the random generator to utilise is also specified, in order to obtain identical results among the diverse implementations of different providers.

4.1.1 Kernel specification

Generate \(2 \cdot N\) pseudo random numbers \(r_j\) with \(1 \leq j \leq 2N\), the numbers will belong to the interval (0,1) while \(N\) is an input parameter of the problem. Then for \(1 \leq j \leq N\) set \(x_j = 2 \cdot r_{2j-1} - 1\) and \(y_j = 2 \cdot r_{2j} - 1\). Thus \(x_j\) and \(y_j\) are uniformly distributed on the interval \((-1,1)\). Next set \(k = 0\). Then beginning with \(j = 1\) test to see if \(t_j = x_j^2 + y_j^2 \leq 1\). If not, reject this pair and proceed to the next \(j\). If this inequality holds, then set \(k = k + 1\), \(X_k = x_j \cdot \sqrt{(-2 \cdot \log(t_j))}/t_j\), \(Y_k = y_j \cdot \sqrt{(-2 \cdot \log(t_j))}/t_j\), where \(\log\) denotes the natural logarithm. Then \(X_k\) and \(Y_k\) are independent Gaussian deviates with mean zero and variance one. Finally, for \(0 \leq l \leq 9\) tabulate \(Q_l\) as the count of the pairs \((X_k, Y_k)\) that lie in the square annulus \(l \leq \max\{|X_k|, |Y_k|\} \leq l + 1\), and output the ten \(Q_l\) counts.
The random sequence \( r_j \) has to be generated through the linear congruential recursion:

\[
    r_j = a \cdot r_{j-1} \pmod{2^{46}}
\]

(4.1)

where \( a \) is the constant 513, and \( r_0 \) is the seed of the sequence, fixed to 281,828,183.

While the above specification is given as stated in [5], a more convenient pseudo-code describing the same problem can be directly derived:

**Listing 4.1: EP Kernel pseudo-code**

```plaintext
// init
SumX = SumY = SquareAnnulus[i] = 0; // for i = 0...9

// perform N iterations
for i = 1 to N
    x = 2 * random() - 1; // x belongs to [-1, 1]
    y = 2 * random() - 1;
    t = x^2 + y^2;
    if (t <= 1) then
        r = ((-2 * log(t)) / t);
        X = x * r; // X is a Gaussian deviate
        Y = y * r;
        SumX += X;
        SumY += Y;
        SquareAnnulus[\max\{|X|, |Y|\}] += 1;
    end if
end

// print computed results
Print SumX, SumY and SquareAnnulus
```

Note that the log function denotes the natural logarithm, while sqrt is the square
root. For this kernel the only input parameter is the amount of iterations to perform, i.e. $N$. For the class C problem, it is fixed $N = 2^{32}$.

Tricks with the aim of avoiding the use of the standard involved functions are explicitly banned [5]. Thus the specification requires to compute the deviates using the built-in functions sqrt and log, as provided by the standard library. Methods such as the table lookup [1] or also the construction of the composite function sqrt(-log(x)) are prohibited. This rule comes as an update of the first specification document (dated 1991), as a consequence that many vendors exploited some unnatural manners to speed up their implementations.

\subsection{Algorithm analysis}

The algorithm is based on the \textit{Marsaglia polar method for normal deviates}, described by Knuth in [25]. The key idea is to generate a random point $P$, with uniform distribution $U$, that lies inside the unit circle. In polar coordinates, the point $P$ is characterised by a radius $R \sim U(0,1)$ and angle $\theta \sim U(0,2\pi)$. Applying the transformations $x \cdot \sqrt{-2 \ln(S)/S}$ and $y \cdot \sqrt{-2 \ln(S)/S}$, where $S = R^2 = x^2 + y^2$,

$x = R \cos(\theta)$ and $y = R \sin(\theta)$, they produce a jointly Gaussian random variable with mean zero and variance one. As this random variable is function of two independent random variables, in each step the method generates a pair of Gaussian outcomes.

On average, the EP algorithm will create $2N \cdot \pi/4$ deviates. It implies that in the above pseudo-code the body of the if command will be executed roughly $N \cdot \pi/4$ times. The “probability of success” $P$ of the condition $x^2 + y^2 \leq 1$ is $\pi/4 \approx 0.78$. The method requires this condition to ensure that the generated random points lie into the unit circle. To derive the probability of success $P$, note that $x$ and $y$ form the set $S = \{x, y \mid -1 \leq x \leq 1 \land -1 \leq y \leq 1\}$. This set describes a square of area 4 centred in the origin. We are filtering from $S$ the set $C = \{x, y \mid x^2 + y^2 \leq 1\}$, which describes a unit circle of area $\pi$. Since the
probability of each point is uniformly distributed, then \( P = \frac{|C|}{|S|} = \pi/4 \).

The complexity of the algorithm is linear with \( N, \Theta(N) \). While it is possible to reorganise the code and to develop an efficient routine to generate the random numbers, improving the overall performance, the cpu spends the greatest amount of time computing the logarithm (log) and the square root (sqrt) functions. The completion time strongly depends by the implementation of these functions, provided by the underlying platform. This scenario is unavoidable, the NPB specification explicitly prohibits the use of other fine tuned software routines that replace the built-in functions.

4.1.3 Implementation

It is more suitable to reorganise the pseudo code of the previous section in distinct phases. The purpose is to reduce potential jumps in the code and promote the vectorisation of the involved computations. The considered phases are:

1. Generate \( 2M \) random numbers, alternately saving the outcomes into the vectors \( R_x, R_y \), each one of dimension \( M \).

2. Compute the vector \( T = 2 \cdot R_x^2 - 1 + 2 \cdot R_y^2 - 1 \).

3. \( \forall T[i], i = 1...M \), test the condition \( T[i] \leq 1 \). If the inequality holds, compute the Gaussian deviates as reported in the previous pseudo-code.

Repeat this cycle until \( N \) iterations are reached, i.e. \( N/M \) times.

It remains to determine the optimal size of \( M \). This value clearly implies the size of the working set. If from one hand, increasing \( M \) decrease the number of jumps, on the other hand a too large working set could involve higher access times due to cache faults. The idea is to fix the size of the working set with the size \( C \) of the cache closer to the cpu. That would bring in the relation \( 3M = C \).
Actually the realised implementation adopts a more heuristic approach. The rationale is that the cache for the first level of the hierarchy has a modest capacity, and the impact of even smaller data structures is not negligible. Moreover nowadays most caches are associative on sets and/or based on rough replacing policies, such as the Not Used Recently (NUR)\textsuperscript{1}. That means that blocks of the current working set could collide even when the cache unit would have additional unused space. These reasons motivate the advantage to underestimate the value of $M$.

The employed heuristic is:

$$M = \frac{1}{3} \cdot (\text{ASSOC} - 1) \cdot \text{LINESIZE} \cdot #\text{GROUPS} \quad (4.2)$$

where \(\text{ASSOC}\) is the associativity of the cache, \(\text{LINESIZE}\) is the size of each block, \(#\text{GROUPS}\) are the total amount of associative groups. This policy entails that the overall occupied space of \(R_x, R_y\) and \(T\) will be \((\text{ASSOC} - 1)/\text{ASSOC}\) of the total cache capacity.

The vectors are implemented as raw C-style arrays. A solution with the standard \texttt{valarray} data structure [31] has also been studied. Although slower, using \texttt{valarrays} the completion time is enough close to the results manually obtained with the C arrays. Nevertheless this problem is rather simple and there is not actually a conspicuous benefit to introduce this abstraction. As mentioned above, several compilers can emit a highly tuned code for C arrays, especially if allocated in the stack and with limited size.

An efficient implementation of the random generator exists for 64-bit architectures. Indeed the computation (4.1) can be directly performed with integers of 8 bytes, without the risk of overflows. The random generator included in the final code is based on the implementation of Cantonnet F, from [10]. NAS authors utilise a similar technique (\texttt{randi8}). When it is more convenient to use integers with less than 8 bytes, such in 32-bit platforms, the same computation needs to

\textsuperscript{1}NUR is an approximation to the Least Recently Used (LRU) policy, where the age of a block is determined by a single bit: recently touched/untouched [12].
be split in more steps, to avoid overflows. The thesis does not treat this scenario.

The parallel algorithm can be straightforwardly derived from the sequential one. There are no shared data structures and there are no inter-iteration dependencies. The only required communication is the sum of the random deviates and of the square annulus at the end of the computation. The parallel algorithm can be obtained splitting the iterations to perform among the available workers. Being \( p \) the parallelism degree, then both of the following approaches are conceivable:

1. Consider macro partitions of size \( N/p \). Each worker will perform the same amount of iterations. This case entails a “map” scenario.

2. Consider partitions of a certain size \( S \). Then schedule the partitions to available workers with an on-demand policy. This case is more similar to a “farm with state” scenario.

The latter approach allows a better load balancing among the workers. The partition size \( S \) should be at least greater than \( 2^{20} \). Experimentally it was noted that with smaller partitions, the overhead, due to the reset of the working sets, increases and could significantly contribute to the overall completion time. However, as the variance on the condition to generate the Gaussian pairs is modest, the former approach can be equally satisfactory [18].

To determine the optimal working set, the code takes into account the same formula for the sequential algorithm. That formula is adapted to the size of the first level of the cache hierarchy. If distinct threads do not share that level, the estimated value for \( M \) will be reasonable. This is not the case if the underlying architecture makes use of a technique of simultaneous multi-threading (SMT), such as Intel Hyper-Threading. However, by default both the operating system and FastFlow schedule, if possible, threads on different CPUs, avoiding the sharing of the first caches.
The generation of the pseudo-random numbers sequence needs some care. The use of an unique random generator for all workers would introduce a potential bottleneck. It is more favourable that each worker would employ a different instance of random generator. Nevertheless, as the final results of the computation must be equal to those observed with the sequential algorithm, the same exact outcomes need to be properly utilised. Whatever the amount of generator instances is, a certain random value must be employed strictly once, while all the numbers of the original pseudo-random sequence must be consumed.

The solution is to associate to each partition the seed to start the sequence of pseudo-random numbers. This operation can be performed during the initialisation. Eventually, when a worker receives a partition, it resets its random generator instance with the seed related to the obtained partition. To jump in the chain of pseudo-random numbers, the code exploits the implementation of J. Burkandt [9], based on the binary algorithm for exponentiation described in [5, 25].

4.2 Kernel CG

The Conjugate Gradient (CG) kernel concerns a “numeric” problem with the use of the common algebraical structures. Involved actions are linear operations and dot products between vectors, matrix - vector multiplications and calculation of norms. The addressed task regards the application of the conjugate gradient method (CGM) to solve a given linear system. The method is reiterated a certain amount of times as tool “to find, through the inverse power method, the smallest eigenvalue of a symmetric definite sparse matrix with random pattern of nonzeros” [5].

The algorithm analysis will concentrate on the matrix - vector multiplication. It is the heavier task and it will be possible to enhance its cost reshaping the structure with a more efficient memory layout. All the other operations feature a fine grain, depending on the result of the previous step. Therefore they are fast
enough to be entirely processed sequentially. On the parallel side, the matrix - vector multiplication can be effectively modelled through a standard data-parallel map paradigm.

4.2.1 Kernel specification

The specification requires to apply a certain amount of times the conjugate gradient for a given application. The dimension $n$ of the vectors, the sparse matrix $A$ and the number of iterations to execute are input parameters of the problem. In the following we will denote scalars with lower case Greek letters, vectors with lower case Latin characters, with $A$ the sparse matrix of order $n$, with $x'$ the transpose of $x$, and with $\| \cdot \|_2$ the norm 2. The following listing reports the code to implement\(^2\):

**Listing 4.2: CG Kernel pseudo code**

```plaintext
INPUT(A, n, niter, \lambda)
set b = [1, ..., 1]'
(start timing here)
FOR it = 1, ..., niter
    Solve with the CGM $Ax = b$ and report $\|r\|$
    $\xi = \lambda + 1/(b'x)$
    print <it, $\|r\|$, $\xi$>
    $b = x/\|x\|$ END FOR
(stop timing here)
```

The conjugate gradient method can be implemented as stated in listing 4.3. This code is equivalent to the one described by the NPB authors in [5].

**Listing 4.3: Implementation of the Conjugate Gradient Method (CGM)**

\(^2\)Compared to the original specification, we have exchanged the letters in order to turn the system into $Ax = b$. In [5] the linear system to treat is $Ay = x$. 39
INPUT(A, b)
set  \(x = [0,\ldots,0]'\)

FOR i = 1, \ldots, 25
  IF (i == 1) THEN // first iteration
    r = p = b;
    \(\rho = r' \cdot r; \quad // = ||r||^2\)
  ELSE
    \(\rho_0 = \rho;\)
    \(r = r - \alpha q;\)
    \(\rho = r' \cdot r; \quad // = ||r||^2\)
    \(\beta = \rho/\rho_0;\)
    \(p = r + \beta p; \quad // descent direction\)
  END IF
  \(q = A \cdot p;\)
  \(\alpha = \rho / (p' \cdot q); \quad // determines the movement\)
  \(x = x + \alpha p; \quad // compute the next solution\)
END FOR

\(q = Ax;\)
\(r = b - q; \quad // residual\)
\(||r|| = \sqrt{r' \cdot r;}\)

return \(x, ||r||;\)

The NPB authors provide a Fortran 77 routine (makea) to generate the sparse matrix \(A\). While its dimension is \(n\), the sparseness degree varies for each kernel class. For the class C, the dimension \(n\) is \(15 \cdot 10^4\). The total number of elements \(M\) turns out to be \(\sim 36 \cdot 10^6\). Note that the matrix is read-only, the algorithm never alters its content. The NPB specification allows to reorganise the matrix to a data structure better suited for the computation. The time spent in the reorganisation does not have to be accounted in the final completion time. The other input parameters are the shift \(\lambda = 110\) and the number of iterations to
perform \text{niter} = 75.

The final output of this algorithm will be the estimation of the eigenvalue \( \xi \). For the class C it must agree with the reference value \( \xi_{\text{ref}} = 28.9736055928 \) with a tolerance of \( 10^{-10} \).

4.2.2 Algorithm analysis

The conjugate gradient is an effective iterative method to solve the linear system \( Ax = b \) when \( A \) is extremely sparse, symmetric and definite positive [30]. The key point is that, for definite positive matrices, the resolution of the equation \( Ax = b \) is equivalent to the problem of minimising its energy \( E(x) = (1/2) \cdot x'Ax - x'b \). At each step the method computes \( x_{i+1} = x_i + \alpha_ip_i \), where \( p \) is the “descent direction” and \( \alpha \) the movement, specified such that \( E(x_{i+1}) < E(x_i) \). In this context, the CG has been employed as the tool to solve the linear system arising from the inverse power method.

In the algorithm, linear operations and dot products between vectors, and matrix - vector multiplications take place. From a theoretical point of view, operations between vectors exhibit a linear complexity. Moreover the distribution of memory accesses is sequential, the operators scan the elements once in linear order. Therefore they are optimal in both aspects. In practice, for the class C kernel, the program spends few seconds to compute all the linear operations between vectors.

The situation is different for the matrix - vector multiplication. It is the heavier task in the algorithm. For the class C, in the machine Andromeda of Figure 5.1, this operation is on the order of \( \sim 100 \) milliseconds. In the whole completion time, that accounts for roughly 99%. Being \( M \) the total number of elements of the matrix \( A \), then the theoretical cost is \( \Theta(M) \). That is unavoidable as each element needs to be processed at least once.

As far as memory layout is concerned, things can vary. The first solution is
to keep in A only the non zero elements ordered per rows. To compute \( y = Ax \), process one row at time calculating \( y_i = A[i,:] \cdot x \) (see Figure 4.1a). If \( r = M/N \) is the average number of elements per row, then each step costs \( O(r) \) random accesses on the vector \( x \). The final cost in the I/O model will be \( O(Nr) = O(M) \) random accesses. In this count, we are not considering the accesses on \( A \) and \( y \) as their distribution is sequential, they have a relatively smaller impact on the total cost.

A different approach is to split the matrix in “column blocks”. Note that the multiplication per columns does not alter the cost. The same amount of random accesses will be on \( y \) instead of \( x \). Nevertheless if we take into account proper blocks of columns, then we will avoid the random jumps on \( y \) (see Figure 4.1b). Let \( B \) be the cache line size, and \( S \) the size interval on \( x \) of the elements indexed in each column block. If \( S \) fits in the cache, a scansion on the block will provoke \( S/B \) random accesses on \( x \). As there are \( N/S \) intervals of \( x \), the whole operation will experience \( O(N/S \cdot S/B) = O(N/B) \) random accesses.

Clearly a proper choice of \( S \) is vital to improve the memory accesses. A too small value would entail more sparse blocks and more random jumps on \( y \). Vice
versa, a too large value would bring back the scenario with the whole multiplication per rows. Given the substantial sparseness of the matrix, it is unlikely that a good value can be decided only taking into account the first cache level. This choice will probably depend on the whole underlying architecture and its cache hierarchy.

Again, for the parallel algorithm it is worth to concentrate just on the matrix multiplication. That can be naturally modelled through the map paradigm. The workers obtain a partition of the matrix at the initialisation time. Eventually, the emitter will send to each worker the vector $x$ to multiply. Finally, the computation encompasses a “reduce” phase, where all the local vectors are combined and summed together. Let $p$ be the given parallelism degree, and $T_{\text{seqmatrix}}$ the sequential completion time of the matrix multiplication, then the latency for this paradigm is:

$$ T_{\text{map}} = T_{\text{multicast}}(N, p) + T_{\text{seqmatrix}}/p + T_{\text{reduce}}(N, p) \quad (4.3) $$

Logically, a pipeline of two processes can describe the whole parallel computation. The former module represents the sequential part of the conjugate gradient method, whereas the latter is the matrix - vector multiplication. The pipeline is cyclic, after a matrix multiplication, the sequential algorithm follows. As there is only a task going back and forth in the graph, the final completion time is given by the sum of the sequential CG method $T_{\text{sequential}}$ with the matrix multiplication $T_{\text{map}}$ times the total number of iterations $niter$ to perform:

$$ T_C = (T_{\text{sequential}} + T_{\text{map}}) \cdot niter \quad (4.4) $$

Figure 4.2 shows the expected completion time for the machine Andromeda and the class C kernel. The sequential times are $T_{\text{sequential}} \approx 1$ ms for the conjugate gradient method (CGM) and $T_{\text{seqmatrix}} = 105$ ms for the matrix - vector multiplication. As the vector $x$ is read-only and the memory space is shared, the multicast
Figure 4.2: Expected completion time for the machine *Andromeda*, described in Figure 5.1a.

operation can just communicate to the workers the involved start/end indices of the original vector. Therefore, its cost can be assumed negligible, $T_{multicast} \approx 0$.

Finally for a linear reduction, $T_{reduce}$ has been evaluated with an upper bound of 2 ms. The rationale is this phase can potentially overlap with the computation of the workers. Since the parallelism degree is limited, the upper bound is given by the time to compute two sums of two vectors of size $N$, located in different memory modules. Nevertheless, for general higher parallelism degrees a better estimation should be determined, as the overlap of the phase with the local computation of the workers will realistically decrease.

### 4.2.3 Implementation

It is a very desirable property to be able to sequentially fetch the matrix from the memory. Section 3.1 shows the benefits that arises from this situation. The matrix
is the largest data structure in the benchmark, it occupies a couple of gigabytes in memory. For what concerns the parallel algorithm, it has been implemented according to the methodology illustrated in section 3.3.2.

It is always possible to arrange the matrix into the memory such that sequential scansions are ensured. This is possible as the matrix is read-only, its content never changes. Moreover the only involved operation is the multiplication by a vector. In memory, the matrix can be represented as an unidimensional sequence of triples as value/column/row. If the values are ordered in some way, then the field regarding the column or the row could be implicit. For instance, if the values are ordered per rows, then it is more profitable to explicitly save only the pairs value/column, keeping the rows apart as pointers to the start / end of the related columns.

The proposed approach assumes that the matrix is ordered by blocks. Each block is subsequently sorted per rows (see Figure 4.1b). As the number of elements in each row varies, the explicit knowledge of this quantity is required and must be saved somewhere. Depending on the dimension $S$ of a partition and being $r$ the average number of elements in a whole row of the matrix, $N$ the vector dimension, then the average number of values per block is $R = r \cdot S/N$. If $R < 1$, then there are empty blocks, jumps on the indexed vectors occur and, generally, the size of the matrix in memory increases.

It is profitable that $R \gg 1$. Otherwise the extra overhead introduced to store the data structure could be significant in the overall computation. Moreover, additional scansions for the columns would take place. Experimentally a good value would seem to be $R = 12 \sim 24$. For the class C kernel, with $N = 150000$ and $r \approx 240$, then $S = 7500 \sim 15000$. The final value strongly depends on the underlying architecture, its cache hierarchy and its access times.

The parallel algorithm has been implemented in FastFlow through the Master/Slave paradigm, as described in section 3.3.2. Figure 4.3 depicts the scheme of the system. All the workers can perform the matrix multiplication. Moreover one of the worker can also execute the sequential part of the CG algorithm. The farm
The Master alternates the execution of a matrix mult, that involves all workers, with the sequential execution of the CG algorithm by the first worker.

The first worker also performs the sequential part of the CG algorithm.

All workers keep a partition of the read-only matrix $A$, and compute the matrix multiplication $y = A \times x$ when demanded by the Master.

Figure 4.3: The scheme adopted in the CG parallel algorithm.

always alternates the matrix multiplication, where all workers are active, with the sequential phase, where only one worker is progressing. Observing that there is only one task in the whole pipeline of the previous section, then this implementation conforms to the described cost model.

4.3 Kernel MG

The MG benchmark simulates the discrete resolution of the equation $\nabla^2 u = v$ by a multigrid method. The algorithm involves operations on grids in the 3D space. It is a recursive method, the computation of each grid relies on the resolution of smaller grids, featuring a V-cycle approach [30]. The grids have periodic boundary conditions, the extreme points border with the values in opposite side. According
to the NPB authors, an objective of this kernel is to “test both short and long distance data communication” [4].

The kernel is rather articulated. From an algorithmic perspective, the main concern is to maximise the use of the vectorisation instructions (SIMD) on the grids. This will result in an interesting case where, through the explicit memory allocation of the grids in a proper way, it is possible to outperform an implementation based on static memory allocation, such as in Fortran 77.

The parallel algorithm can be modelled by a static stencil pattern. The workers will own a different partition of the main grid and both its direct and indirect projections. To update their partitions, they need to exchange the boundaries with their neighbours. In our model, a boundary is a single depth of the 3D grid. Therefore each worker will present two neighbours. This choice will entail a ring structure for the linkage of the workers. Finally, once the grids become suitably smalls, the algorithm will continue sequentially.

4.3.1 Kernel specification

The kernel requires the implementation of the V-cycle multigrid iterative method, to resolve the linear system $Au = v$. The maximum grid size $N$ and the number of iterations to perform $N_{ITER}$ are input parameters of the problem. It is a recursive algorithm, the total number of levels is $K = \log_2(N)$. In the following we will denote with lower case letters the grids and with capital letters the involved grid operators. The notation $x_k$ represents the grid $x$ for the level $k$. The listing 4.4, based on [1], outlines the iterative algorithm to implement$^3$:

```
Listing 4.4: Serial MG algorithm

INPUT( N , NITER , v )
K = log2(N)
```

$^3$ That is equivalent to the original and recursive definition given in the NPB specification [5], but it is simpler to analyse and more efficient to implement.
set $z_K = 0$

FOR $i = 1, \ldots, \text{NITER}$

// evaluate the residual on the finest grid
$r_K = v - Az_K$

// down cycle
FOR $k = K, K-1, \ldots, 1$
    $r_k = Pr_{k+1}$ // projection
END FOR

// smoothing on the coarsest grid
$z_1 = Sr_1$

// up cycle
FOR $k = 2, \ldots, K-1$
    $z_k = Qz_{k-1}$ // interpolation
    $r_k = r_k - Az_k$ // residual
    $z_k = z_k + Sr_k$ // smoothing
END FOR

// last iteration
$z_K = z_K + Qz_{K-1}$ // add interpolation
$r_K = v - Az_K$ // error
$z_K = z_K + Sr_K$ // smoothing

END FOR

// $z_K$ is the final solution, $r_K$ the error.
return $||r_K||_2$
Figure 4.4: The relative distance for the operator coefficients. Consider a generic point at the row $i$, column $j$ and depth $k$. The table in b) depicts the 2D grid for the fixed depth $k$ centred on the black element $i,j$. The numbers represent the distance of the cell from $i,j,k$. Thus in b), the numbers labelled with 1 are the values that differ by one only on the row or on the column. Similarly, numbers labelled with 2 differ by one in both column and row, but the depth is the same. Instead a) is the 2D grid centred on $i,j$ but with the depth $k-1$. The distance of the values is equal to the corresponding points in b) +1, as they also differ on the depth. A similar reasoning also applies to c), that depicts the 2D grid for the depth $k+1$.

The capital letters A, P, Q, S represent the residual, projection, interpolation and smoothing operators, respectively. The operators are defined in terms of the coefficients applied to the point $<i, j, k>$ and its adjacent values. The coefficient $c_d$ applies to the points that are at distance $d$ from the central point $i, j, k$. The distance $d$ is obtained differing by one exactly any $d$ of the three indices $i, j, k$, see Figure 4.4.

The residual and smoothing operators do not alter the dimensions of the applied grid. The projection operator, acting on a grid of dimensions $N$, retrieves a grid of size $N/2$ for each of its dimensions, i.e. $N/8$ times smaller. Vice versa, the interpolation operator Q doubles the size of each dimension of the applied grid.

For the class C kernel, the system deals with the resolution of the grid with dimensions $N = 512 = 2^9$. The number of iterations is NITER = 20. The grid $v$ is composed by zeros, except in 20 predetermined points. The table 4.1 shows

---

$^4$Actually A denotes the discretisation of the Laplacian $\nabla^2$ operator. We name it residual operator to emphasize that it is always applied to evaluate the residual on the current solution.
the coefficients for the involved operators. The verification test is performed on
the norm 2 of the last residual, that must agree with a given reference value. The
timing must start after the initialisation of $r_K$ and $v$ and before the first evaluation
of the residual. The simulation must stop the timer after the computation of the
norm 2, but before displaying or printing its value.

4.3.2 Algorithm analysis

Multigrid methods are another possibility to efficiently resolve a linear system
$Au = v$, with $A$ big, sparse and symmetric [30]. These methods still exploit
the classical, stationary iterative solvers, named *smoothers* in this context, but on
different grids and levels. The key idea is to reduce the residual $v - Au$ obtained by
the smoother in a smaller, coarser, faster and easier grid. In the V-cycle approach,
the method recursively further reduce the grid, applies the smoother on the coarse
grid and adds the correction back to the finer grids. Eventually this process boosts
the convergence speed of the iterative solver.

The main part of this section concerns the computational cost of the multigrid
operators. Differently from the other kernels, We will not employ the asymptotic
algorithmic notation big O, as the hidden constants do play an important role on
the final cost. We will take into account the arithmetic cost, i.e. the number of
additions and multiplications that an operator has to execute. Going forward, this
section deals with the boundary conditions related to the grid. Finally, it describes
the parallel algorithm, characterised by a stencil paradigm.

<table>
<thead>
<tr>
<th>Operator</th>
<th>$c_0$</th>
<th>$c_1$</th>
<th>$c_2$</th>
<th>$c_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual (A)</td>
<td>-8/3</td>
<td>0</td>
<td>1/6</td>
<td>1/12</td>
</tr>
<tr>
<td>Projection (P)</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
<td>1/16</td>
</tr>
<tr>
<td>Interpolation (Q)</td>
<td>1</td>
<td>1/2</td>
<td>1/4</td>
<td>1/8</td>
</tr>
<tr>
<td>Smoothing (S)</td>
<td>-3/17</td>
<td>1/33</td>
<td>-1/61</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 4.1: Multigrid operator coefficients for the class C kernel*
Applying the operator on each isolated point is rather ineffective. As the final result will depend on 27 values, the computation of a single point requires 27 additions and 4 multiplications, one per coefficient. Since each grid contains \( N^3 \) points, the final cost of this approach is \( 27 \cdot N^3 \) additions and \( 4 \cdot N^3 \) multiplications. This procedure is also ineffective from the point of view of the memory accesses. Due to the high number of required sparse points, the distribution of accesses could potentially appear random to the processor.

Another weakness for the \textit{naive} implementation is that it does not really take into account the vectorisation of the arithmetic expressions. Current x86_64 processors may execute two (SSE) or four (AVX) additions within a single instruction on two contiguous and non overlapping memory chunks [19]. It is convenient to fix the 3D grid representation on the unidimensional memory space. We order the points by depth, row and column. Thus, two consecutive values in the column dimension are also adjacent in the memory representation. Two values that differ only by one row, have distance \( N \) in the memory space, and if the differ only by one depth, they have distance \( N^2 \).

Our goal will be to reduce the number of required additions and to promote the potential use of SIMD instructions emitted by the compiler. To achieve that, the solution is to \textit{precompute} some values for each row. Let \( \Gamma \) be the argument grid, \( G \) the resulting grid, \( T_1 \) and \( T_2 \) two supporting arrays of size \( N \). Fix the current depth \( \hat{k} \) and row \( \hat{i} \), then compute the values:

\[
\forall j = 1, ..., N \quad T_1[j] = \Gamma(\hat{i}, j, \hat{k} - 1) + \Gamma(\hat{i}, j, \hat{k} + 1) + \Gamma(\hat{i} + 1, j, \hat{k}) + \Gamma(\hat{i} - 1, j, \hat{k}) \\
T_2[j] = \Gamma(\hat{i} + 1, j, \hat{k} - 1) + \Gamma(\hat{i} - 1, j, \hat{k} - 1) + \Gamma(\hat{i} + 1, j, \hat{k} + 1) + \Gamma(\hat{i} - 1, j, \hat{k} + 1)
\]  

The values of \( G \) for the fixed depth \( \hat{k} \) and row \( \hat{i} \) can be computed (see Figure 4.4)
as:

\[
\forall j = 1, \ldots, N \quad G(\hat{i}, j, \hat{k}) = c_0 \cdot \Gamma(\hat{i}, j, \hat{k}) \\
+ c_1 \cdot (\Gamma(\hat{i}, j - 1, \hat{k}) + \Gamma(\hat{i}, j + 1, \hat{k}) + T_1(j)) \\
+ c_2 \cdot (T_1(j - 1) + T_1(j + 1) + T_2(j)) \\
+ c_3 \cdot (T_2(j - 1) + T_2(j + 1))
\]

The complexity for this scheme is bounded by \(16 \cdot N\) additions and \(4 \cdot N\) multiplications per row. More importantly, the expressions 4.5 and 4.6 can be easily vectorised by the compiler, as they scan different memory chunks. For what concerns the I/O model, the same expressions sequentially access the involved rows. If the working set \(12 \cdot N\) fits into the data cache, then this scheme will cost \(12 \cdot N/B\) memory accesses per row in (almost) sequential manner. Even if a L1 cache cannot keep all the working set for \(N = 512\), it will stay into L2 for sure. The above analysis applies for the general worst case scenario.

Actually, operators may exploit the fact that some coefficient is zero (see Table 4.1). In the residual, the coefficient \(c_1\) is zero, so the second line of the expression 4.7 can be removed. The number of operations per row drops to \(13N\) additions and \(3N\) multiplications. Similarly, for the smoothing operator, the coefficient \(c_3\) is zero. Therefore this operator undergoes \(14N\) additions and \(3N\) multiplications per row. To obtain the whole cost of the operator, multiply the values for the total number of rows, i.e. \(N^2\).

When the expression to compute is of the type \(G = G \pm OP(\Gamma)\), there is an extra addition per element in the total cost. That occurs for all the smoothing operations, but the one on the coarsest level, and the residuals, but the one on the finest level. The cost of the smoothing on the first level is negligible, it works on a grid of 8 values. The cost of the finest residual \(r_K = v - Az_K\) is actually equivalent to the cost of \(r_K = Az_K\). Indeed as \(v\) contains only 20 non zeroes values, we can compute the whole expression as \(r_K = -Az_K\), and eventually add \(v\).
The projection is apparently the more expensive operator. It is applied on a grid $\Gamma$ of dimension $2N \times 2N \times 2N$. Adapting the generic strategy delineated above, this operator would undergo $8 \cdot 2N + 8 \cdot N = 24N$ additions per row, while the multiplications remain equal. Nevertheless, the algorithm cannot execute a projection in the finest grid, that is 8 times bigger than the grid on the level below. That means we have not to pay the cost of this operator for the most expensive level.

The interpolation operator will employ a different scheme. Actually the operator aims to perform a trilinear interpolation [30]. We need to distinguish between even and odd columns, rows and depths. The value for odd columns / rows / depths depends on the linear interpolation of the adjacent even columns / rows / depths. For even rows $i$, columns $j$, depths $k$ in $G$, the resulting value is the corresponding column $i/2$, row $j/2$ and/or depth $k/2$ in $\Gamma$.

In the following scheme we will move by steps on $\Gamma$. A single depth $k$ of $\Gamma$ will allow to compute the values of the depths $2k$ and $2k + 1$ of $G$. Similarly, a row $i$ of $\Gamma$ will allow to calculate the values $2i$ and $2i + 1$ of $G$. $\Gamma$ has dimension $M^3$ while $G$ is $2M^3 = N^3$. As for the previous generic scheme, it is convenient to precompute some values of $\Gamma$ for each row $i$ and depth $k$:

$$\forall j = 1, \ldots, M \quad T_1[j] = \Gamma(\hat{i}, j, \hat{k}) + \Gamma(\hat{i} + 1, j, \hat{k})$$
$$T_2[j] = \Gamma(\hat{i}, j, \hat{k} + 1) + \Gamma(\hat{i} + 1, j, \hat{k} + 1) + T_1[j]$$

$T_1$ represents the sum of the current row $\hat{i}$ and of the subsequent row $\hat{i} + 1$, while $T_2$ is the sum of the adjacent rows in the current depth $\hat{k}$ and in the next depth $\hat{k} + 1$. The cost of the precomputation is 3 additions per element, that is $3 \cdot M$ additions per row. Now we will consider the four separate cases with even depth /
Expression          Adds     Mults     Description
-------------------  --------  --------  -------------------------------
$r_K = v - A z_K$    $13 N^3$   $3 N^3$   Residual on the finest grid
$r_k = P r_{k+1}$   $24 N^3$   $4 N^3$   Projection on coarser grids
$z_k = Q z_{k-1}$   $5/4 N^3$   $N^3$     Interpolation on coarse grids
$r_k = r_k - A z_k$ $14 N^3$   $3 N^3$   Subtract the residual on coarse grids
$z_k = z_k + S r_k$ $15 N^3$   $3 N^3$   Smoothing
$z_K = z_k + Q z_{K-1}$ $9/4 N^3$   $N^3$     Interpolation on the finest grid

Table 4.2: Arithmetic cost of the multigrid operators

The total cost, including the precomputation, is $10 \cdot M$ additions and $8 \cdot M$ multiplications per row. There are $M^2$ rows in $\Gamma$, therefore the whole cost of the interpolation is $10 \cdot M^3$ adds and $8 \cdot M^3$ multiplications. As $M = N/2$, that cost eventually becomes $5/4 \cdot N$ adds and $N$ mults per row, in terms of $G$.

Table 4.2 sums up the cost of each operator. The residual and the smoothing experience almost the same number of arithmetic operations. These are the most expensive tasks of the multigrid algorithm. The cost of a projection on a level $l$ is comparable to the cost of an interpolation on the level $l + 1$. Their added cost is roughly half of the cost of the residual or smoothing in the same level $l$. 

\begin{align*}
    \forall j = 1, \ldots, M 
    & G(2\hat{i}, 2j, 2\hat{k}) = c_0 \cdot \Gamma(\hat{i}, j, \hat{k}) \quad (4.10) \\
    & G(2\hat{i}, 2j + 1, 2\hat{k}) = c_1 \cdot (\Gamma(\hat{i}, j, \hat{k}) + \Gamma(\hat{i}, j + 1, \hat{k})) \\
    & G(2\hat{i} + 1, 2j, 2\hat{k}) = c_1 \cdot T_1(j) \quad (4.11) \\
    & G(2\hat{i}, 2j, 2\hat{k} + 1) = c_1 \cdot (\Gamma(\hat{i}, j, \hat{k}) + \Gamma(\hat{i}, j, \hat{k} + 1)) \\
    & G(2\hat{i}, 2j + 1, 2\hat{k} + 1) = c_2 \cdot (\Gamma(\hat{i}, j, \hat{k}) + \Gamma(\hat{i}, j, \hat{k} + 1) \\
    & \quad + \Gamma(\hat{i}, j + 1, \hat{k}) + \Gamma(\hat{i}, j + 1, \hat{k} + 1)) \\
    & G(2\hat{i} + 1, 2j, 2\hat{k} + 1) = c_2 \cdot T_2(j) \quad (4.12) \\
    & G(2\hat{i} + 1, 2j + 1, 2\hat{k} + 1) = c_2 \cdot (T_2(j) + T_2(j + 1)) \\
    & G(2\hat{i} + 1, 2j, 2\hat{k} + 1) = c_3 \cdot T_2(j) \\
    & G(2\hat{i} + 1, 2j + 1, 2\hat{k} + 1) = c_3 \cdot (T_2(j) + T_2(j + 1)) \quad (4.13)
\end{align*}
The periodic boundaries are the extreme points on every dimensions. Values preceding the first column are on the last column. Likewise, the values succeeding to the last column are on the first column. The same reasoning applies for points on the border of the other dimensions. To deal with this property, a viable solution, advised in [1], is to add a series of “ghost” points on the border of the dimensions. These additional points replicate the content of the opposite side of the dimension, avoiding the necessity to jump back and forth in the memory to retrieve their values.

The scheme presented to compute the operators needs to be slightly adjusted. Dimensions have size $N + 2$, instead of $N$. The idea is to calculate only the real values during the application of the operator, and to replicate the ghost points at the end. Expressions 4.8 and 4.9 rely on arrays of size $N + 2$, and iterate $j$ from 0 to $N + 1$. The core calculus 4.7 remains untouched. The interpolation is an exception. Due to the different cases that the algorithm has to take into account, it is probably more suitable to explicitly perform the interpolation on the ghost points for the rows and the columns, but copying the last two depths from the beginning of the grid. The overall complexity of the operators does not practically change.

The parallel algorithm can be modelled by a static stencil pattern. Each worker keeps a partition of both the residual $r_K$ and the correction $z_K$, together with their direct and indirect projections. In our model the atomic unit to potentially split a grid is a single depth. That means for a grid of dimension $N^3$ there can be up to $N$ partitions, which it is acceptable for both the class C kernel and the maximum parallel degree reachable in the considered simulations. The computation evolves in steps where the involved operator is independently applied on each partition, and the boundaries of the result are exchanged between neighbours at the end. As each worker keeps two neighbours, one for the upper and one for the lower boundary depth, the final topology of the stencil is a ring.

Going down in the levels, the partitions become smaller. At some point, the
computation can continue sequentially. A special worker is promoted to perform the sequential part of the algorithm for the lower levels. This node will *gather* all the active partitions from the other workers for a properly predetermined *cutoff* threshold level. This behaviour will occur in the down cycle of listing 4.4, originating a serial projection. Eventually the worker will resume the stencil *scattering* the computed grid in the up cycle to the available workers, which will execute an interpolation. Experimentally, a suitable value for the *cutoff* would seem 4 or 5. The sequential worker will take a bunch of milliseconds to perform all the required computation.

A cost model can be obtained summing up the cost of each required task. Being \( p \) the parallelism degree, each task in the stencil will cost \( T_{seqop}(l) + 2 \cdot T_{send}(l) \), depending on the sequential time of the operation \( op \) and on the involved level \( l \). The total cost will be:

\[
T_{op}(l) = \frac{T_{seqop}(l)}{p} + 2 \cdot T_{send}(l) \tag{4.14}
\]

\[
T_{multigrid} = NITER \cdot \left[ T_{sequential} + T_{resid}(\log_2(N)) \right. \\
+ \left. \sum_{k=cutoff+1}^{\log_2(N)} \left( T_{project}(k) + T_{interp}(k) + T_{resid}(k) + T_{smooth}(k) \right) \right] \tag{4.15}
\]

Figure 4.5 shows the predicted completion time for the machine *Andromeda* (see

<table>
<thead>
<tr>
<th>Level</th>
<th>Residual</th>
<th>Smoothing</th>
<th>Interpolation</th>
<th>Projection</th>
<th>Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>714</td>
<td>766</td>
<td>206</td>
<td>n/a</td>
<td>1.3</td>
</tr>
<tr>
<td>8</td>
<td>78</td>
<td>86</td>
<td>25</td>
<td>217</td>
<td>0.5</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>11</td>
<td>3.3</td>
<td>25</td>
<td>( \approx 0 )</td>
</tr>
<tr>
<td>6</td>
<td>1.2</td>
<td>1.3</td>
<td>0.4</td>
<td>3.5</td>
<td>( \approx 0 )</td>
</tr>
<tr>
<td>( \leq 5 )</td>
<td>( \approx 0 )</td>
<td>( \approx 0 )</td>
<td>( \approx 0 )</td>
<td>( \approx 0 )</td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.5: Expected completion time for the machine *Andromeda* with the kernel MG on class C

Chap. 5, Figure 5.1), according to the above cost model. Table 4.3 reports the times, expressed in milliseconds, to instantiate the model. The cut-off threshold is fixed at level 5. The serial computation $T_{\text{sequential}}$ has been weighted with 5 ms.

### 4.3.3 Implementation

In this benchmark, the vectorisation by SIMD instructions plays a crucial role. All the optimisation techniques described in section 3.2 are fundamental. The parallel algorithm has been mapped to a peculiar Master/Slave module.

It is possible to boost the execution of the arithmetic expressions through a proper alignment of the grid in memory. As the algorithm operates by rows, these have to be memory aligned to fully take advantage of the vectorisation. Similarly
the arrays $T_1$ and $T_2$ should be allocated with a proper memory alignment. In the
developed implementation, these “vectors” will be aligned by the cache line size
$CL$, that is usually of 64 bytes. Finally, all involved pointers should be declared
both restricted and aligned.

The grid must be allocated taking into account some padding among the rows.
Note that, while the theoretical grid dimension $N$ is a power of 2, the actual
dimension will be $N + 2$, due to ghost points at the boundaries. If the grid was
allocated as a single chunk of memory of $(N + 2)^3$ continuous values, then the rows
would be inevitably misaligned. Considering the padding, the memory size of a
grid is given by:

$$S_{\text{row}} = \left\lceil \frac{N + 2}{CL / \text{sizeof(double)}} \right\rceil \cdot CL$$  \hspace{1cm} (4.16)

$$S_{\text{depth}} = S_{\text{row}} \cdot (N + 2)$$  \hspace{1cm} (4.17)

$$S_{\text{grid}} = S_{\text{depth}} \cdot (N + 2) = S_{\text{row}} \cdot (N + 2)^2$$  \hspace{1cm} (4.18)

The term $S_{\text{grid}}$ determines the size of an entire grid $G$ of dimension $N^3$. In the
parallel algorithm, given the parallelism degree $p$, each worker will encompass a
partition of size $(M + 2) \cdot S_{\text{depth}}$, where $M = N/p$. A value at depth $k$, row $i$ and
column $j$ will be mapped at the position $k \cdot S_{\text{depth}} + i \cdot S_{\text{row}} + j$.

This solution dramatically exploits the vectorisation for every row. If the AVX
instruction set is available in the underlying architecture, then the expressions 4.5
and 4.6 may be computed within a single addition per iteration. Even with the
SSE instruction set, this part will be processed quickly. Expression 4.7 is more
ticklish. It mixes additions and multiplications in a not enough regular manner.
Some vectorisation may occur, but in a more limited perspective.

As described in section 3.3.2, a peculiar Master/Slave pattern implements the
parallel algorithm. Figure 4.6 depicts the scheme of the system. The workers perform the different stages of the stencil computation, coordinated by the master unit. A special worker is promoted to carry on the sequential part as well, under the threshold cut-off, of the algorithm. This scheme is common to all implemented kernels and can be applied without major differences also in this scenario.
Chapter 5

Experiments

We run a full set of experiments with the three kernels described in the previous chapter. The tested programs are the derived implementation in C++/FastFlow and the reference Fortran 77 / OpenMP code, provided by the NAS authors [22]. The developed code is publicly available, under the GNU GPL license, at: http://www.dadamax.net/downloads/thesis.tar.gz. Figure 5.1 shows the configuration of the two multi-core machines taken into account.

Both the machines present two interconnected Intel Xeon nodes, featuring a NUMA topology. The caches are of type exclusive in the first two levels, and shared in each node for the third level. Andromeda supports the SSE2 instruction set, while Pianosa both SSE2 and AVX. The reference NAS implementation is v3.3, last release up to the current date [26]. The employed FastFlow library is version 2.0, taking the revision 44 from the SVN development branch [16]. In Andromeda, there were some issues with that version, so the rev.40 was instead used. The implementation uses the C++ Boost libraries [8] merely for configuration purposes.

The experiments do not take advantage of Simultaneous Multi-Threading (SMT). The two machines make use of Intel Hyper-Threading technology [13]. When Hyper-Threading is active, the cores replicate the state registers to handle two
## Figure 5.1: Configuration of the machines for the conducted experiments

<table>
<thead>
<tr>
<th></th>
<th>a) Andromeda (andromeda.di.unipi.it)</th>
<th>b) Pianosa (pianosa.di.unipi.it)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor model:</strong></td>
<td>Two interconnected Intel Xeon E5520 @ 2.27 Ghz</td>
<td>Two interconnected Intel Xeon E5-2650 @ 2 Ghz</td>
</tr>
<tr>
<td><strong>Cores, threads:</strong></td>
<td>8 cores, 16 threads</td>
<td>16 cores, 32 threads</td>
</tr>
<tr>
<td><strong>Caches:</strong></td>
<td>L1: 32 Kb, L2: 256 Kb, L3: 8 Mb (shared per node)</td>
<td>L1: 32 Kb, L2: 256 Kb, L3: 20 Mb (shared per node)</td>
</tr>
<tr>
<td><strong>Memory:</strong></td>
<td>(\approx 12) Gb</td>
<td>(\approx 32) Gb</td>
</tr>
<tr>
<td><strong>Operating system:</strong></td>
<td>Linux CentOS 5.7 on kernel 2.6.18 /64 bit</td>
<td>Linux CentOS 6.3 on kernel 2.6.32 /64 bit</td>
</tr>
<tr>
<td><strong>Compiler:</strong></td>
<td>GCC 4.7.2, libc 2.5</td>
<td>GCC 4.7.2, libc 2.12</td>
</tr>
<tr>
<td></td>
<td>C++ Boost library 1.52</td>
<td>C++ Boost library 1.52</td>
</tr>
</tbody>
</table>

Threads simultaneously. Active threads on a single core compete for the usage of the physical resources. The final goal is to improve the global utilisation of a single core. However, there is no gain for computational optimised applications, particularly with an extreme high usage of the memory bandwidth. Besides that, measuring the benefits of SMT is beyond the scope of the thesis.

The achievable parallelism degree depends on the number \(N\) of available cores. That is \(N = 8\) for Andromeda, and \(N = 16\) for Pianosa. In the FastFlow implementation, the maximum achievable parallelism degree is actually \(N - 1\), to account the service node, the Master, see section 3.3.2. A logical thread is always
univocally mapped to a single core.

All programs were compiled by GCC 4.7. This version of the compiler introduces particular constructs to improve automatic vectorisation, see section 3.2. Common optimisation flags, -03, -march=native and -ffast-math, were enabled for both programs. To compile the F77/OpenMP codes, the flag -mcpu=medium is further required, due to static memory size constraints. Refer to the compiler manual [21] for an overview of the meaning and the entailments that these flags yield.

Each test has been run five times. Reported results rely on the observed median. The accounted measures are the completion time $T_C$, the scalability and the speed up. Complying with [11], the scalability is defined as the ratio of the execution times between the parallel algorithm with parallelism degree 1 and the corresponding algorithm with parallelism degree n, i.e. $T_{par}(1)/T_{par}(n)$. The speed up is the ratio between the best sequential algorithm and the parallel algorithm, i.e. $T_{seq}/T_{par}(n)$. In this context, $T_{seq}$ is equal for both programs and it will be the sequential execution time of the C++ or F77 implementation, whatever is the fastest.

5.1 Kernel EP

Kernel EP is implemented using a standard farm parallel design pattern. As it should not undergo particular overheads, the scalability should be close to the ideal. The sequential completion times were 302 secs on Andromeda and 341 secs on Pianosa for the custom C++ implementation, and 306 secs on Andromeda and 332 secs on Pianosa for the Fortran 77 code, provided by the NAS authors. Therefore, the sequential algorithms in C++ and F77 experience similar execution times. Figure 5.2 shows the observed completion times.

In theory, the performance of the two implementations should be similar. The
weight of the sequential part of the computation is practically the same. The size of the partitions is big enough such that workers may proceed uninterruptedly for long times, while the involved communications are relatively scarce. In the graphs, the completion times overlap almost everywhere. The gain of an additional computational unit in F77 / OpenMP is limited. For the FastFlow outcomes, the measured standard deviation is bounded by $\leq 100$ milliseconds. Indeed, in F77 / OpenMP the standard deviation reached a couple of seconds.

Figure 5.3 shows the measured scalability (top) and speed up (bottom). On Andromeda, the performance of FastFlow with parallelism degree $p = 1$ is somehow poor. However, the problem does not affect higher parallelism degrees. This result distorts the significance for the scalability. On Pianosa, the performance of the sequential algorithm and of the parallel algorithm with $p = 1$ is practically equivalent. The scalability and speed up graphs mainly coincide.

In conclusion, the speed up of both implementations is very close to the ideal one. On Andromeda, the speed up measured for OpenMP is lower than FastFlow. Nevertheless, reaching $p = 8$, OpenMP achieves a (slightly) better completion time.
in absolute terms. On Pianosa, speed up and scalability are optimal. Surprisingly, OpenMP does not improve with $p = 15$, obtaining the same speed up with $p = 16$ as FastFlow with $p = 15$.

## 5.2 Kernel CG

Kernel CG presents a Map + Reduce pattern. The algorithms perform the matrix - vector multiplication using two different approaches. The Fortran 77 code multiplies the matrix by rows, while the derived implementation accomplishes the operation by “column blocks”, due to memory access reasons. For the custom implementation, the sequential algorithm completed in $\sim 208$ secs on Andromeda and $\sim 175$ secs on Pianosa. The Fortran 77 code took $\sim 259$ secs (+ 24%) on
Andromeda and \( \sim 350 \text{ secs} \) (+100\%) on Pianosa, underpinning the rationale of the formulated analysis.

Figures 5.4 and 5.5 report the observed completion times for the parallel algorithms on Andromeda and Pianosa, respectively. The expected time comes up from the model developed in section 4.2.2. Given the parallelism degree \( p \), on Andromeda, the OpenMP implementation is between the 25\% and 50\% slower than the FastFlow counterpart. On Pianosa, the completion time of OpenMP is beyond the 40\% higher than FastFlow, except for the last degrees, where it is at least 20 \% higher.

Figure 5.6 shows the measured scalability and speed up for both algorithms. On Andromeda, OpenMP scales worse than FastFlow, while the situation is the opposite on Pianosa. In this machine, the f77 implementation is 2x slower than the developed C++ code. Therefore, the margin of improvement is greater. The
Figure 5.5: Kernel CG: recorded completion times for Pianosa

speed up depicts the actual gain of each implementation respect to a common reference, that is in absolute terms.

The model does not properly match the implementation. It is tight only with $p \leq 4$ on Andromeda, and $p \leq 7$ on Pianosa. This is likely a consequence of the rough estimation of the reduce phase, in constant terms. Going up with the parallelism degree, the computational grain of the map stage decreases, while the number of workers increase. Thus, the implicit pipeline effect between the map and the reduce goes away, and workers begin to perform the reduce phase in strict sequence. To get a closer approximation, the model should account the reduce as function of the parallelism degree.

Summing up, there is an effective improvement of the derived implementation compared to reference F77 code. The algorithms behind the C++ and F77 implementations are different. Section 4.2.2 described the potential beneficial to split the computation by proper blocks. The result was the sequential C++ algorithm being
up to 2x faster than the original one. Considering the parallel scenario, OpenMP scales lesser than FastFlow on Andromeda, and better on Pianosa. Nevertheless, OpenMP should have recovered a conspicuous margin. In absolute terms, the final speed up turned out (at least) the 25% smaller.
Figure 5.6: Observed scalability and speed up of the algorithms for the kernel CG. The abscissa is the parallelism degree, while the ordinate is the measured scalability (a,b) or speed up (c,d), depending on the graph.
5.3 Kernel MG

Kernel MG presents a static stencil pattern. The derived implementation is particularly optimised. The main difference with the F77 / OpenMP is the extra padding to favour the vectorisation instructions. The sequential algorithm completed on Andromeda, exploiting the SSE2 vectorisations, in ≈ 58 secs, and on Pianosa, with the AVX instruction set, in ≈ 52 seconds. The F77 implementation took ≈ 78 secs (+34%) on Andromeda, and ≈ 58 secs (+13%) on Pianosa. The discrepancy in the gain is likely a consequence that the AVX instruction set is less strict than SSE2 for the alignment requirements [19].

The figures 5.7 and 5.8 depict the observed completion time for the parallel algorithms. In general, the C++ / FastFlow is the 10% faster than the reference F77 / OpenMP code. The divergence increases to 50% on Andromeda for \( p \leq 6 \), but the implementation does not scale with \( p = 7 \). On Pianosa, the difference eventually becomes less than 1 second, for an algorithm that completes in ≈ 7.5

![Andromeda - Completion time](image)

Figure 5.7: kernel MG, recorded completion times for Andromeda
Figure 5.8: kernel MG, recorded completion times for Pianosa.

seconds. The model, developed in section 4.3.2, does not match the actual trend of the measured execution times.

Both programs suffer of severe scalability issues. The algorithm demands a large effort in terms of memory bandwidth, due to the size of the grids, up to the order of gigabytes, and the nature of operators, that simultaneously act on distant values in a non cache friendly way. The evaluated machines consist of two NUMA nodes. Generally, the derived implementation, similarly to the behaviour of OpenMP, maps the physical threads interleaving the deployment onto one node to the other. The purpose is to double the overall available memory bandwidth and the caches capacity, as they are crucial resources.

Figure 5.9 reports the measured scalability mapping all workers to a single node of Pianosa. As mentioned above, this is not the strategy of the actual tests, but it shows how much a node is able to scale for the evaluated algorithm. In this case, there is no communication among the workers, as the data structure resides
Figure 5.9: kernel MG, achieved scalability when mapping all workers into a single core, in Pianosa.

into the same node. The threads share the cache on the third level, the main memory and the memory bus. Eventually, they saturate the memory bandwidth, causing the stalling of the processor pipeline. The graph resembles this aspect for the higher parallelism degrees, where the scalability does not improve any further.

The model does not take into account the memory usage of the computation. In theory, the algorithm should be able to reach near optimal scalability. The model starts from the execution times of the sequential algorithm, and predicts the trend dividing the times by the amount of available workers. The key point is, since the memory bandwidth per worker is lower than the sequential case, the actual computation time could be worse than $T_{seq}/p$, especially in memory intensive applications. This is an intrinsic problem of multicore systems.

Considering both NUMA nodes, we can observe a kind of “stairs” effect. When $p$ is odd, i.e. $\exists n \in \mathbb{N} : p = 2n + 1$, then the scalability augments proportionally as in the single node system with $\bar{p} = n$. When $p$ is even, the scalability improves, as the partitions in the two nodes are balanced. Indeed, with $p$ odd, any one of the two nodes must keep at least one additional worker. This entails that, a node
must process more than $> 1/2$ of the shared data structure. While, with $p$ even, the two nodes demand the same amount of memory bandwidth.

Figure 5.10 shows the measured scalability and speed up. The reported trend complies with the rationale outlined above. In general, the multigrid method is an effective algorithm that should efficaciously rendered in parallel, more than the conjugate gradient. Nevertheless, in a multicore architecture, the memory bandwidth represents an issue that could severely limit its scalability. Due to extreme memory bandwidth requirements, the derived algorithm for the kernel MG is likely more suitable for a distributed system.
Figure 5.10: Observed scalability and speed up of the algorithms for the kernel MG. The abscissa is the parallelism degree, while the ordinate is the measured scalability (a,b) or speed up (c,d), depending on the graph.
Chapter 6

Conclusions

The thesis examined a set of the NPB algorithms for the multicore architecture. It mainly pursued two directions. The former was a thorough analysis of the selected kernels. The latter was an effective methodology to describe the algorithms with the skeletal approach supported by the FastFlow framework. To accomplish the goals, standard techniques were exploited, relying on algorithmic models, cache friendliness, well known C++ optimisations and structured parallel programming. The final experiments showed several achievements, compared to the reference provided by the NPB authors.

The derived C++ implementation sustained a fair competition with the F77 programs. Some previous works [14, 20] pointed out the performance penalties of porting the Fortran 77 reference codes in other languages, which caused a starting handicap in their tests. The thesis described the major differences between C++ and F77 and explained the necessary solutions to plug the gap. At the end, the implemented C++ programs were able to achieve the same execution times, without sacrificing much of the expressiveness that F77 does not permit.

Actually, the thesis also introduced some improvements in the sequential algorithms. In kernel EP, it proposed a scheme to fully exploit the cache capacities.
In kernel CG, the thesis devised a more convenient procedure to perform the multiplication of the sparse matrix, formally justified by the model. The result was a difference of up to 2x on *Pianosa*, respect to the original method. In kernel MG, the thesis discussed the additional enhancements given by the explicit treatment of automatic vectorisation. Though this was a predominantly numeric code, the derived implementation turned out to be faster of 10% – 30% than the Fortran 77 reference.

The C++/FastFlow program showed a better speed up than F77/OpenMP codes. Even when the scalability was superior in OpenMP, as in Figure 5.6b, the measured speed up was equal or inferior. Indeed, the scalability is a relative quantity. Sometimes, it can be misleading as a program that starts from a slower sequential code, it has greater margins of improvement. The speed up is instead an absolute measure, as it entails a common reference. A higher speed up always means a faster program. Clearly, in the final result, there is the summed contribution of algorithmic enhancements, optimisations, the C++ language and the FastFlow framework.

Summing up, the main objective of the thesis was to develop an efficient implementation of a subset of the NPB kernels. Starting from the NPB specification, the thesis studied the computational aspects of the selected problems. It delineated a common methodology, describing how to take advantage of the memory, the necessary optimisations and the strategy to efficiently parallelise the algorithms. Then, it covered each kernel one by one, applying the common methodology to overcome the critical aspects reckoned. The comparison tests assessed the worth of the derived implementation.

The implementation was explicitly tuned up for the underlying platform. While the Fortran 77 / OpenMP implementation aims to attain very general portability too, this was not an objective of the thesis. FastFlow runs on top of Linux/Unix, some introduced optimisations leverage on the characteristic of the x86/64 architecture, while the custom implementation used the Linux SysFS to understand
the topology of the nodes and the nature of the caches. Note that NPB codes are strongly optimised as well, as the compiler may take advantage of the complete and static knowledge of both the Fortran 77 program and the OpenMP directives.

Multicore systems are an efficient solution to deploy small scale applications. The main advantages are the ease of development and the small overhead in communication and synchronisation, compared to distributed systems. When more cores are hosted into the same node, scalability issues can appear due to the saturation of the available memory bandwidth. The kernel MG was an instance of this problem. It is not simple to predict and to model the weight of the memory in a computation, as it depends on the actual distribution of the accesses. In these cases, designing memory efficient algorithms becomes essential.

The approach sustained by FastFlow may be beneficial. FastFlow is naturally oriented towards computations on streams. For data-parallel scenarios, such as the CG and MG kernels, a developer should adapt its computation very carefully. The mapping of the computational units to the physical threads is opportune, and a developer should address this aspect as well. In contrast, OpenMP is easier and more direct to handle. While C++ and FastFlow enable an additional flexibility, that can be favourable when a developer is aware how to exploit it.

Currently, FastFlow is under active development from the University of Pisa. Experimental tests, to support and integrate clusters of machines, distributed systems and gpus, are ongoing. The project is ambitious. The goal is to offer a common set of abstractions, such as the structured parallel patterns exploited in the thesis, to express practical problems, reducing the possible incurred overheads, in a multiple variety of environments. The benchmarks developed in the thesis served as a baseline to describe concrete applications and verify its performance versus other standards, in this specific case OpenMP.
Appendix A

Source code

The section shows the derived implementation in C++ / FastFlow. It reports the code for the three kernels, sequential and parallel, and the involved data structures. The developed program actually encloses additional components for configuration and optimisation reasons. As they do not concern the definition of the algorithms, they are not shown. The whole code is available at\(^1\): http://www.dadamax.net/downloads/thesis.tar.gz. In particular, these are the additional components that are not covered here:

- Base classes: provide the kernel input parameters and a common abstraction to invoke the algorithms.
- Configuration: loads at runtime and holds the configuration parameters.
- Cpu optimisation: defines architecture dependent macros, such as aligned pointers (see section 3.2.3) and cache sizes.
- Cpu topology: parses the cache structure through the SysFS interface [12], and infers the physical threads suitable for pinning.
- Main: launches the computation according to the parameters specified by the user.
- Random: the pseudo-random generator, consistent with the NPB specification [5], and based on the implementation of Cantonnet et al. [10] and Burkardt [9].
- Spin Lock: a wrapper of the spin lock mutex, provided by the FastFlow library.
- Timer: a commodity wrapper for `sys/time.h`.

\(^1\)Released under the GNU GPL license.
Listing A.1: EP Sequential algorithm

```c
// local variables
double local_sum_x(0), local_sum_y(0);
uint64_t local_square_annulus[10] = { 0 };
Random random(seed); // random generator
// space for the computed random numbers
// M is a proper value based on the heuristic formula 4.2
double random_numbers[M];
double temps[M / 2]; // holds t = x^2 + y^2
// amount of steps to perform
uint64_t step_iterations = 2 * (iterations - 1) / M + 1;
// avoids an additional conversion
uint64_t uM = static_cast<uint64_t>(M);

// warm the cache
for (uint64_t i = 0; i < M; i++)
    random_numbers[i] = 0;
for (uint64_t i = 0; i < M / 2; i++)
    temps[i] = 0;

// start the timer
timer.start();

for (uint64_t i = 0; i < step_iterations; ++i) {
    // compute in advance the random numbers
    uint64_t step_sz = (i < step_iterations - 1) ? uM : 2 * iterations - i * uM;
    for (uint64_t j = 0; j < step_sz; ++j) {
        random_numbers[j] = random.next() * 2.0 - 1.0;
    }
    step_sz /= 2;

    // compute t = x^2 + y^2
    for (uint64_t j = 0; j < step_sz; ++j) {
        temps[j] = pow(random_numbers[2 * j], 2.0) + pow(random_numbers[2 * j + 1], 2.0);
    }

    // generate the random deviates
    for (uint64_t j = 0; j < step_sz; ++j) {
        if (temps[j] > 1) continue; // slower than the combo while+if, but cleaner
        double & t = temps[j];
        double & x = random_numbers[2 * j];
        double & y = random_numbers[2 * j + 1];
        double r = sqrt((-2.0 * log(t)) / t);
        double X = x * r;
        double Y = y * r;
        local_sum_x += X;
        local_sum_y += Y;
        local_square_annulus[static_cast<unsigned int>(j)] += X * X + Y * Y;
    }
}
```
To represent the parallel computation, the code describes a farm with state. It is necessary to create both the Emitter and the set of Workers. These are instance members of the custom class Parallel. Listing A.2 shows the derived interface for the wrapper and the inner classes Emitter and Worker. A mutex is required to merge, at the end of the computation, the local state of each worker with the global counters in the Base class. A task is the association between a seed and the number of iterations to perform.

The computation carried by Workers resembles the above sequential algorithm. The initialisation phase is moved into the constructor. The arrays random_numbers and temps are not allocated into the stack, but onto the heap. To avoid the aliasing issue, they are declared restricted by the svc method. Listing A.3 reports the implementation of the parallel algorithm.

---

**Listing A.2: EP Parallel algorithm interface**

```cpp
class Parallel: public Base {
private:
    typedef std::pair<uint64_t, uint64_t> task_t;
    typedef boost::mutex mutex_t;
    typedef boost::lock_guard<mutex_t> lock_t;
    typedef std::vector<ff::ff_node*> workers_t;

    /∗ ∗ Description of the farm emitter ∗ /
    /*
    class Emitter: public ff::ff_node {
    private:
        uint64_t* seeds;
        std::size_t seeds_size; // size of the array seeds
        uint64_t task_iterations; // amount of iterations for intermediate seeds
        uint64_t task_iterations_last; // amount of iterations for the last seed
        std::size_t index; // current index
    public:
        /∗ ∗
        * Constructor
        * @param wrapper: the instance of the Parallel wrapper.
        /∗ /
        Emitter(Parallel* wrapper);
        /∗ ∗
        * Destructor
        /∗ /
        virtual ~Emitter();
        /∗ ∗
        * contract — Emits the next task to the workers
        /∗ /
        void* svc(void*);
```
 class Worker: public ff::ff_node {

private:
    static uint64_t worker_id_counter; // id counter
    // object id
    uint64_t worker_id;
    // local sums of the X and Y random deviates
    double worker_sum_x, worker_sum_y;
    // local computed square annulus
    uint64_t worker_square_annulus[10];
    // step size, iterations, etc.
    std::size_t M; // worker double step size
    uint64_t uM; // avoids an additional conversion
    double* worker_random_numbers;
    my::Random random; // random generator
    double* worker_temps; // holds t = x^2 + y^2
    // related ParallelEP instance
    Parallel* ep_instance;

public:

    /**
     * Default constructor
     */
    Worker(Parallel* wrapper);

    /**
     * Destructor
     */
    ~Worker();

    /**
     * Perform the computation of the random deviates for sequence
     * instructed in the task
     * @param task involved partition, the type must be std::pair*
     * @return ff::GO_ON
     */
    void* svc(void* task);

    /**
     * Copies local results to the relative ParallelEP instance.
     */
    void svc_end();

};

// friendship relationships with the inner classes
friend class Emitter;
friend class Worker;

// farm objects
ff::ff_farm<> farm;
Emitter emitter;
workers_t workers;

// mutex to protect the access from the workers to instance variables
mutex_t mutex;

protected:

// contract— it runs the computation
void execute();

// Resets the initial values for the internal variables
void touch();

public:

/**
 * Constructor
 */
Parallel();

/**
 * Destructor
 */
virtual ~Parallel();

Listing A.3: EP Parallel implementation

// Constructor
Parallel::Parallel() : Base(), farm(), emitter(this), workers(), mutex() {
  // number of workers
  std::size_t parallelism_grade = get_configuration().get_parallelism_grade();

  // Scheduling on demand (round robin)
  farm.set_scheduling_ondemand();

  farm.add_emitter(&emitter);

  workers.reserve(parallelism_grade); // sets the capacity of the vector
  for (std::size_t i = 0; i < parallelism_grade; i++)
    workers.push_back(new Worker(this));
  farm.add_workers(workers);
}

// Destructor
Parallel::~Parallel() {
  // delete workers
  std::size_t size = workers.size();
  for (std::size_t i = 0; i < size; i++) {
    delete ((Worker*) workers[i]);
    workers[i] = NULL;
  }
}

// Runs the algorithm
void Parallel::execute() {
    timer.start();
    int status = farm.run_and_wait_end();
    timer.stop();
    if (status < 0) {
        throw KernelException("[ParallelEP::execute] bad status: " + str(status));
    }
}

void Parallel::touch() {
    sum_x = 0;
    sum_y = 0;
    for (std::size_t i = 0; i < square_annulus_size; i++) square_annulus[i] = 0;
    timer.reset();
}

// Reset the internal state

Parallel::Emitter::Emitter(Parallel* wrapper) {
    Configuration& config = get_configuration();
    std::string strategy = config.exists(OPT_SCH_STRATEGY) ? config.get<std::string>(OPT_SCH_STRATEGY) : DEF_SCH_STRATEGY;

    // The user can specify the amount of tasks to emit.
    // Parse the configuration to retrieve the number and the size of partitions
    // (...)
    task_iterations = wrapper->iterations / partitions;
    task_iterations_last = task_iterations + (wrapper->iterations % partitions);
    seeds = Random::fill(wrapper->seed, partitions, 2 * task_iterations); // init task_iterations ahead!!!
    seeds_size = partitions;
    index = 0;
}

Parallel::Emitter::~Emitter() {
    if (seeds != NULL) { delete[] seeds; seeds = NULL; }
}

void* Parallel::Emitter::svc(void*) {
    task_t* task(NULL);
    if (index < seeds_size - 1) {
        task = new task_t(seeds[index], task_iterations);
    } else if (index == seeds_size - 1) {
        task = new task_t(seeds[index], task_iterations_last);
    } else { // end
        task = NULL;
    }
}
index++;  
return task;
}  

/**************************************************************************  
  WORKER  
**************************************************************************/

uint64_t Parallel::Worker::worker_id_counter = 0;

Parallel::Worker::Worker(Parallel* wrapper) :  
worker_id(Worker::worker_id_counter++), worker_sum_x(), worker_sum_y(),  
worker_square_annulus(), random(0), ep_instance(wrapper) {  
Configuration& c = get_configuration();  

// sets the step size, iterations, etc.
M = c.get(OPT_DOUBLE_STEP_SIZE, static_cast<std::size_t >(((2./3) *  
opt_double_cfirst_heurystic_size)));  
if (M % 2 == 1) M++; // the double step size must be multiple of 2
uM = static_cast<uint64_t>(M);

// allocate worker_random_numbers
// opt_cache_level1_linesize is the linesize of the cache L1
int result = posix_memalign((void**) &worker_random_numbers,  
opt_cache_level1_linesize, M * sizeof(double));  
if(result!=0){ /* throws an exception */  
}

// allocate worker_temps
result = posix_memalign((void**) &worker_temps, opt_cache_level1_linesize,  
M * sizeof(double) /2);  
if(result!=0){ /* throws an exception */  
}
}

Parallel::Worker::~Worker() {  
// delete worker_random_numbers;
free(worker_random_numbers); worker_random_numbers = NULL;
// delete worker_temps
free(worker_temps); worker_temps = NULL;
}

void* Parallel::Worker::svc(void* task_raw) {
  task_t* task = (task_t*) task_raw;  
uint64_t seed = task->first;
uint64_t iterations = task->second;
delete task;

  // restricted aliases  
double* restrict worker_random_numbers = CACHEAligned(this->  
worker_random_numbers);
double* restrict worker_temps = CACHEAligned(this->worker_temps);
uint64_t restrict worker_square_annulus = this->worker_square_annulus;
random.set_seed(seed); // sets the initial seed for the random generator
uint64_t step_iterations = 2 * (iterations - 1) / uM + 1; // amount of  
steps to perform

  for (uint64_t i = 0; i < step_iterations; ++i) {  

// compute in advance the random numbers
uint64_t current_step_size =
  (i < step_iterations - 1) ? uM : 2 * iterations - i * uM;
for (std::size_t j = 0; j < current_step_size; ++j) {
  worker_random_numbers[j] = random.next() * 2.0 - 1.0;
}

current_step_size /= 2;

// compute t = x^2 + y^2
for (std::size_t j = 0; j < current_step_size; ++j) {
  worker_temps[j] = pow(worker_random_numbers[2 * j], 2.0)
    + pow(worker_random_numbers[2 * j + 1], 2.0);
}

// generate the random deviates
for (std::size_t j = 0; j < current_step_size; ++j) {
  if (worker_temps[j] > 1)
    continue; // slower than the combo while + if j, but cleaner
  double& t = worker_temps[j];
  double& x = worker_random_numbers[2 * j];
  double& y = worker_random_numbers[2 * j + 1];
  double r = sqrt((-2.0 * log(t)) / t);
  double X = x * r;
  double Y = y * r;
  worker_sum_x += X;
  worker_sum_y += Y;
  worker_square_annulus[static_cast<unsigned int>(std::max(fabs(X),
    fabs(Y)))++] +=
}

// return /*ff::*/GO_ON; // complaining gcc 4.6.3
return GO_ON ;

void Parallel::Worker::svc_end() {
  lock_t lock(ep_instance->mutex);
  ep_instance->sum_x += worker_sum_x;
  ep_instance->sum_y += worker_sum_y;
  for (int i = 0; i < 10; i++)
    ep_instance->square_annulus[i] += worker_square_annulus[i];
}
Kernel CG

Sequential algorithm

Listing A.4: CG / sequential.h

```cpp
1 class Sequential : public Base{
2     private:
3         // size of the arrays / order of the vectors
4         const unsigned N;
5
6         // vectors
7         double* b; // samples
8         double* x; // current solution
9         double* r; // residuals
10         double* p; // direction
11         double* q; // support vector, q = A * p
12
13         // the sparse matrix A
14         Matrix A;
15
16         // the amount of iterations to perform for the Conjugate Gradient method
17         const unsigned cg_iterations;
18
19         // performs an iteration of the inversion power method
20         // and returns the current value for zeta
21         private: double inverse_power_method(unsigned it);
22
23         // performs the conjugate gradient method with the current x,b vectors
24         // and returns the norm 2 of r = ||r|| = || b - Ax ||
25         private: double conjugate_gradient();
26
27         // perform a free (not timed) iteration of the inversion power method
28         protected: virtual void touch();
29
30         // perform the actual algorithm
31         protected: virtual void execute();
32
33 public:
34     /**
35         * Constructor
36         */
37         Sequential();
38
39     /**
40         * Destructor
41         */
42         virtual ~Sequential();
43 }
```

Listing A.5: CG / sequential.cpp

```cpp
1 // Constructor
2 Sequential::Sequential() : Base(), N(input_N),
3     // init the matrix
4     A(input_N, input_nonzer, input_lambda, get_configuration().get("sequential.row_block_size",0u), get_configuration().get("sequential.column_block_size",0u)),
5     // amount of iterations
```
cg_iterations(get_configuration().get<unsigned>("cgiterations"))
{
  // destroy the makea wrapper, initialised by A
  MakeaWrapper::remove();

  // create the vectors
  b = new double[N];
  x = new double[N];
  r = new double[N];
  p = new double[N];
  q = new double[N];
}

// Destructor
Sequential::~Sequential() {
  delete[] b; b = NULL;
  delete[] x; x = NULL;
  delete[] r; r = NULL;
  delete[] p; p = NULL;
  delete[] q; q = NULL;
}

// Reset the internal state and performs a fake iteration
void Sequential::touch(){
  for(unsigned i = 0; i < N; i++) { b[i] = 1; }
  inverse_power_method(0);
}

// Executes the sequential algorithm
void Sequential::execute(){
  // restore b = [1 ... 1]'
  for(unsigned i = 0; i < N; i++) { b[i] = 1; }
  double zeta(0);

  timer.start();
  for(unsigned it = 1; it <= input_niter; it++){
    zeta = inverse_power_method(it);
  }
  timer.stop();
  output_z = zeta;
}

double Sequential::inverse_power_method(unsigned it){
  // execute the conjugate gradient to solve A x = b
  double rnorm = conjugate_gradient();

  // compute zeta = shift + 1/(b\times x)
  double bx = 0;
  for(unsigned j = 0; j < N; j++) { bx += b[j] * x[j]; }
  double zeta = input_lambda + 1 / bx;

  // print the current iteration, \|r\| and zeta
  if(it>0) { printf("it: %d, \|r\|: %.15f, zeta: %.15f\n", it, rnorm, zeta); }

  // compute the norm2 of x, \|x\|
  double xnorm = 0;
  for(unsigned j = 0; j < N; j++) { xnorm += pow(x[j], 2); }
}

\[ \text{xnorm} = \sqrt{\text{xnorm}}; \]

// update b = x / ||x||
for(unsigned j = 0; j < N; j++) { b[j] = x[j] / xnorm; }

return zeta;
}

double Sequential::conjugate_gradient(){
    const unsigned n(N); // size of the vectors
    double alpha; // movement
    double rho = 0; // direction
    double pAp; // = p' A p

    // reset x = [0, ..., 0]
    memset(x, 0, n * sizeof(double));

    // body iterations
    for(unsigned i = 1; i <= cg_iterations; i++) {
        if(i == 1) { // first iteration
            for(unsigned j = 0; j < n; j++) {
                r[j] = p[j] = b[j];
                rho += pow(b[j], 2);
            }
        } else {
            double rho0 = rho;
            // r = r - alpha * q
            for(unsigned j = 0; j < n; j++) { r[j] = r[j] - alpha * q[j]; }
            // rho = r' * r
            rho = 0;
            for(unsigned j = 0; j < n; j++) { rho += pow(r[j], 2); }
            double beta = rho / rho0;
            // p = r + beta * p
            for(unsigned j = 0; j < n; j++) { p[j] = r[j] + beta * p[j]; }
        }

        // q = A * p
        A.multiply(p, q);
        pAp = 0; // p' * q = p'A q
        for(unsigned j = 0; j < n; j++) { pAp += p[j] * q[j]; }
        alpha = rho / pAp; // = rho / p'q = rho / p'A p

        // next solution
        for(unsigned j = 0; j < n; j++) { x[j] += alpha * p[j]; }
        // printf("it: %d, rho: %f, alpha: %f\n", i, rho, alpha);
    }

    // q = A x
    A.multiply(x, q);

    // r = b - A x
    for(unsigned j = 0; j < n; j++) { r[j] = b[j] - q[j]; }
}
Matrix definition

Listing A.6: CG / matrix.h

```c
/**
 * Data structure to represent the sparse matrix employed in the kernel CG.
 * The matrix can either hold the whole CG matrix or rectangular partitions of
 * a certain size. The only operation provided is the multiplication with a
 * vector, performed by blocks of the size specified in the constructor.
 * The data structure is not thread safe.
 */
class Matrix {
  protected:
    unsigned index_first_row; // the first index for this partition
    unsigned index_first_column; // the first column for this partition
    unsigned number_rows; // number of contiguous rows represented in the
                           // matrix
    unsigned number_columns; // number of columns in the matrix
    unsigned number_blocks; // total count of blocks in the matrix
    unsigned size_values; // number of non zero values
    unsigned size_row_block; // the size of each row block
    unsigned size_col_block; // the size of each column block
    unsigned num_row_blocks; // number of row blocks that form the matrix
    unsigned num_column_blocks; // number of column blocks that form the matrix

    // array to track the number of non zero values in each block
    unsigned* elements_in_the_block;

    // column indices of related values
    // the first value in a block of columns is the number of following columns
    // values in the same block
    unsigned* columns;

    // non zero values of this matrix
    double* values;

  public:
    /**
     * Constructor.
     * Creates the sparse matrix invoking the routine makea, developed by NPB
     * authors.
     * @param n order of the matrix, parameter of the kernel/class
     * @param nonzer coefficient of non zero elements, parameter of the
     * kernel/class
     * @param shift lambda coefficient, parameter of the kernel/class
     * @param blocksize the size of each block of the matrix. By default
     */
};
```
Matrix(unsigned n, unsigned nonzer, double shift, unsigned row_block_size = 0, unsigned col_block_size = 0, unsigned first_row = 0, unsigned num_rows = 0, unsigned first_column = 0, unsigned num_columns = 0);

/**
 * Default destructor
 */
virtual ~Matrix();

/**
 * It multiplies this matrix by y and stores the result in x, i.e. y = Ax.
 * It is assumed that x and y have the same dimension of the matrix.
 * The memory addresses of both x and y vectors must not overlap.
 * @param x input vector
 * @param y output vector
 */
void multiply(double* x, double* y);

---

Listing A.7: CG / matrix.cpp

```cpp
#include <algorithm> // max
#include <stdexcept>
#include "matrix.h"
#include "makea_wrapper.h"

#ifndef restrict
#define restrict __restrict
#endif

Matrix::Matrix(unsigned n_, unsigned nonzer_, double shift_, unsigned rowblocks_, unsigned colblocks_, unsigned first_row_, unsigned num_rows_,
    unsigned first_column_, unsigned num_columns_) {
    // check args
    if(first_row_ >= n_) { /* throws an exception */ }
    if(num_rows_ == 0) num_rows_ = n_; 
    if(first_row_ + num_rows_ > n_) num_rows_ = n_ - first_row_; 
    if(first_column_ >= n_) { /* throws an exception */ }
    if(num_columns_ == 0) num_columns_ = n_; 
    if(first_column_ + num_columns_ >= n_) num_columns_ = n_ - first_column_; 
    if(rowblocks_ == 0 || rowblocks_ > num_rows_) rowblocks_ = num_rows_; 
    if(colblocks_ == 0 || colblocks_ > num_columns_) colblocks_ = num_columns_; 

    // makea is the fortran 77 routine to generate the input matrix, provided
    // by the NPB authors. MakeaWrapper, not shown here, is a C++ bridge to
    // invoke the F77 routine. The parameters are the same of the f77 original
```
MakeaWrapper& makea = MakeaWrapper::get_instance(n_, nonzer_, shift_);

// define the dimensions of the involved blocks
index_first_row = first_row_;  
index_first_column = first_column_;  
size_row_block = rowblocks_;  
size_col_block = colblocks_;  
number_rows = num_rows_;  
number_columns = num_columns_;  
size_values = makea.rowstr[index_first_row + number_rows] - makea.rowstr[index_first_row];
unsigned blocks_per_row = number_columns / size_col_block;  
if((number_columns % size_col_block) > 0) blocks_per_row++;  
number_blocks = blocks_per_row * number_rows;

// create the matrix
columns = new unsigned[size_values];  
values = new double[size_values];  
elements_in_the_block = new unsigned[number_blocks];
unsigned index_value = 0;  
unsigned index_block = 0;  
unsigned f77indices[size_row_block];  // last column indices

// row and column blocks
num_row_blocks = number_rows / size_row_block;  
if(number_rows%size_row_block>0) num_row_blocks++;  
num_column_blocks = number_columns / size_col_block;  
if((number_columns%size_col_block)>0) num_column_blocks++;  

// move by row blocks
for(unsigned I = 0; I < num_row_blocks; I++){  
  unsigned row_block_start = index_first_row + I * size_row_block;  
  unsigned row_block_end = (I < num_row_blocks -1) ?  
    row_block_start + size_row_block : (index_first_row + number_rows);  

  // move by column blocks
  for(unsigned J = 0; J < num_column_blocks; J++){  
    unsigned col_block_start = index_first_column + J * size_col_block;  
    unsigned col_block_end = J < num_column_blocks -1 ?  
      col_block_start + size_col_block :  
      (index_first_column + number_columns);  

    // move by single rows
    unsigned f77indextrack=0;  // current position in the f77indices
    for(unsigned row = row_block_start; row < row_block_end; row++){  
      // starting column value
      unsigned f77index_start = makea.rowstr[row] -1;  
      // ending column value
      unsigned f77index_last = makea.rowstr[row +1] -2;  
      unsigned& f77k = f77indices[f77indextrack++];  
      // track the amount of columns in this block
      unsigned index_c0 = index_value;

      // first column block?
      if(index_column_block == 0){  
        f77k = f77index_start;  
        // skip rows that are outside the column index
        while(f77k <= f77index_last & & makea.colidx[f77k] -1 <  
          static_cast<int>(index_first_column)) f77k++;  
    }
while(f77k <= f77index_last &&
     makea.colidx[f77k] -1 < static_cast<int>(col_block_end))
{
    values[index_value] = makea.values[f77k];
    columns[index_value] = makea.colidx[f77k] -1;
    index_value++;
    f77k++;
}

elements_in_the_block[index_block++] = index_value - index_c0;
}
} // end for (single rows)
} // end for (col blocks)
} // end for (row blocks)

/**
 * realloc to match the actual amount of collected non zero values, that
 * can be lesser to what retrieved from the makea array, as this partition
 * may have skipped some values out of the column bounds.
 */
if(size_values > index_value){
    size_values = index_value;
    double* values_old = values;
    unsigned* columns_old = columns;
    double* values_new = new double[size_values];
    unsigned* columns_new = new unsigned[size_values];
    memcpy(values_new, values_old, size_values* sizeof(double));
    memcpy(columns_new, columns_old, size_values* sizeof(unsigned));
    delete[] values; values_old = 0;
    // values_old confuses valgrind
    delete[] columns; columns_old = 0;
    values = values_new;
    columns = columns_new;
}
}
} // Destructor
Matrix::~Matrix() {
    delete[] elements_in_the_block; elements_in_the_block=0;
    delete[] columns; columns=0;
    delete[] values; values=0;
}

// Multiply the matrix by the vector x and store the result in y, i.e. y = A x.
void Matrix::multiply(double* restrict x, double* restrict y){
    unsigned index_value = 0;
    unsigned index_block = 0;

    // disable aliasing
    unsigned* restrict elements_in_the_block = this->elements_in_the_block;
    unsigned* restrict columns = this->columns;
    double* restrict values = this->values;

    // move by row blocks
    for(unsigned I = 0; I < num_row_blocks; I++){
        unsigned row_block_start = index_first_row + I * size_row_block;
        unsigned row_block_end = (I < num_row_blocks - 1) ?
            row_block_start + size_row_block : (index_first_row + number_rows);
        } // end for (single rows)
} // end for (col blocks)
} // end for (row blocks)

/**
 * realloc to match the actual amount of collected non zero values, that
 * can be lesser to what retrieved from the makea array, as this partition
 * may have skipped some values out of the column bounds.
 */
if(size_values > index_value){
    size_values = index_value;
    double* values_old = values;
    unsigned* columns_old = columns;
    double* values_new = new double[size_values];
    unsigned* columns_new = new unsigned[size_values];
    memcpy(values_new, values_old, size_values* sizeof(double));
    memcpy(columns_new, columns_old, size_values* sizeof(unsigned));
    delete[] values; values_old = 0;
    // values_old confuses valgrind
    delete[] columns; columns_old = 0;
    values = values_new;
    columns = columns_new;
}
}
} // Destructor
Matrix::~Matrix() {
    delete[] elements_in_the_block; elements_in_the_block=0;
    delete[] columns; columns=0;
    delete[] values; values=0;
}

// Multiply the matrix by the vector x and store the result in y, i.e. y = A x.
void Matrix::multiply(double* restrict x, double* restrict y){
    unsigned index_value = 0;
    unsigned index_block = 0;

    // disable aliasing
    unsigned* restrict elements_in_the_block = this->elements_in_the_block;
    unsigned* restrict columns = this->columns;
    double* restrict values = this->values;

    // move by row blocks
    for(unsigned I = 0; I < num_row_blocks; I++){
        unsigned row_block_start = index_first_row + I * size_row_block;
        unsigned row_block_end = (I < num_row_blocks - 1) ?
            row_block_start + size_row_block : (index_first_row + number_rows);
Parallel algorithm

The following code shows the implementation of the parallel algorithm. The class Parallel describes a farm with a custom Emitter (the Master) and custom Workers (the Slaves). The code achieves the synchronisation between the master and the slaves through an atomic counter, current_active_workers, that determines the number of active workers. The counter is reset by the Master and decreased by the Slaves when ending their piece of computation.

As described in section 4.2.2, the master and the slaves alternate two states. All the workers execute the matrix multiplication, while a special worker, named SuperWorker, is responsible to also perform the sequential part of the conjugate gradient method. The TextStatus determines the current state in the Master node, consisting of the above two stages and the phases of initialisation and termination. To select the proper worker in the sequential phase, the code provides a custom scheduler.

Listing A.8: CG / parallel.h

```cpp
// move by column blocks
for(unsigned J = 0; J < num_column_blocks; J++){
    // move by rows
    for(unsigned row = row_block_start; row < row_block_end; row++){
        // for the first column block, set to zero the value in y
        if(J == 0){ y[row] = 0; }

        // move by columns
        double sum = 0;
        const unsigned size = elements_in_the_block[index_block++];
        for(unsigned i = 0; i < size; i++, index_value++){
            sum += values[index_value] * x[columns[index_value]];
        }
        y[row] += sum;
    } // end for (rows)
} // end for (column blocks)
} // end for (row blocks)
```

```cpp
// classes defined in this header
class Parallel;
class Scheduler;
class Task;
class Emitter;
class Worker;

/**
 * Shared state between the Master and the "Slaves"
 */
enum TaskStatus{ TASK_STATUS_INIT, TASK_STATUS_CGKERNEL,
                 TASK_STATUS_MATRIX_MULT, TASK_STATUS_END };

class Parallel: public my::cg::Base {
```
protected:
  // Container for the workers
  typedef std::vector<ff::ff_node*> workers_t;
  workers_t workers;
  int num_workers;

  // Farm
  ff::ff_farm<Scheduler>* farm;
  Emitter* emitter;

  // Synchronise master/workers
  TaskStatus current_status;
  Task* current_task;
  atomic_t current_active_workers;

  // set the final output of this kernel
  protected: virtual void set_output(double zeta);

  // perform a free (not timed) iteration of the inverse power method
  protected: virtual void touch();

  // perform the actual algorithm
  protected: virtual void execute();

public:
  /**
   * Constructor
   *
   * Parallel();
   */

  /**
   * Default destructor
   *
   * virtual ~Parallel();
   */

  friend class SuperWorker;
};

/**
 * Distinguish the type of generated task
 *
 * enum Task_Operation{ TASKInicialisation, TASK_multiplication };
 */

/**
 * The payload sent from the Emitter to the Workers
 *
 * struct Task{
 *   Task_Operation op;

 *   Task(Task_Operation op_): op(op_){ }
 *   virtual ~Task(){ }
 * }
 */

/**
 * Defines the parameters of each worker's partition.
 *
 * struct Task_Initialisation : public Task{
 *   unsigned n;
 *   unsigned nonzer;
 *   double shift;
 */

93
unsigned rowblocks;
unsigned colblocks;
unsigned first_row;
unsigned num_rows;
unsigned first_column;
unsigned num_columns;

Task_Initialisation() : Task(TASK_INITIALISATION){
    n = nonzer = shift = rowblocks = colblocks = first_row = num_rows =
    first_column = num_columns = 0;
}

/**
 * Dictates to workers to execute a multiplication
 */
struct Task_Multiplication: public Task{
    double* x;
    double* y;
    Task_Multiplication(double* x_=0, double* y_=0) : Task(TASK_MULTIPLICATION)
        , x(x_), y(y_){}
};

/**
 * Scheduler of the tasks among the workers.
 * If pick.cgmaster.next() has been invoked, then it will return as next worker
 * the node implementing the sequential CG kernel.
 * Otherwise it returns the workers according to the round-robin policy.
 */
class Scheduler: public ff::ff_loadbalancer{
protected:
    int num_workers; // number of available workers = parallelism grade
    int node_cgkernel; // id of the worker that implements the cg
    // sequential kernel
    int next_worker; // the next worker id to return
    bool flag_pick_master; // flag to select the super worker

    /**
     * Contract — select the next worker according to the scheduling policy
     */
    inline int selectworker(){
        if(flag_pick_master){
            next_worker = 0;
            flag_pick_master = false;
        return node_cgkernel;
        } else { // round robin
            int current_worker = next_worker;
        next_worker = (next_worker + 1) % num_workers;
        return current_worker;
        }
    }

public:
    /**
     * Constructor
     * @param num_workers: the number of available workers
     * @param node_cgkernel: id of the master worker that implements the CG sequential kernel
     */
Scheduler(int num_workers_, int node_cgkernel_ = 0) : ff_loadbalancer(num_workers_), num_workers(num_workers_), node_cgkernel(node_cgkernel_), next_worker(0), flag_pick_master(false) { }

/**
 * Sets the number of available workers
 */
void set_size_workers(int num){ num_workers = num; }

/**
 * Sets the master worker that implements the CG sequential kernel
 */
void set_cgmaster(int node_id){ node_cgkernel = node_id; }

/**
 * Asks to return the next time the master worker
 */
void pick_cgmaster_next(){ flag_pick_master = true; }
}; // class Scheduler

/**
 * Emitter module for the underlying farm paradigm
 */
class Emitter: public ff::ff_node{
protected:
    const int cpu_id; // cpu where to pin the emitter
    const int num_workers; // total number of workers
    TaskStatus& status; // current status
    atomic_t& waiting_for_tasks; // track how many workers are still working
    Task*& task_args; // task args
    Scheduler& scheduler; // reference to the farm scheduler

public:
    /**
     * Constructor
     */
    Emitter(int cpu_id, int num_workers, TaskStatus& status, atomic_t& completed_tasks, Task*& task_ptr, Scheduler& scheduler);

    /**
     * Pins the emitter to the specified cpu_id
     */
    virtual int svc_init();

    /**
     * Contract — implements the logic of the emitter
     */
    virtual void* svc(void*);
};

/**
 * A nominal worker of the farm, only able to perform the matrix multiplication
 */
class Worker: public ff::ff_node{
protected:
typedef SpinLock mutex_t;
static mutex_t mutex;

const int cpu_id; // cpu identifier where this worker has been pinned
Matrix* matrix_part; // partition of the matrix
double* local_y; // local vector y to perform the matrix mult.
int local_N; // the dimension of the vector y
TaskStatus& status; // synch with the current state of the Emitter
atomic_t& waiting_for_tasks; // counter for active workers

virtual void initialise(Task_Initialisation* task);

virtual void worker_cgkernel(Task_Multiplication* task_mult);

void worker_multiply(double* x, double* y);

public:

virtual ~Worker();

virtual int svc_init();

virtual void* svc(void*);

};

A worker that also executes the sequential computation of the algorithm

class SuperWorker: public Worker{
protected:
    Parallel& parallel; // reference to the Parallel wrapper;

const unsigned N; // size of the arrays / order of the vectors
const unsigned niter; // amount of iterations to perform
const double lambda; // zeta shift
const unsigned cg_iterations; // number of iterations for the CGM

// vectors
double* b; // samples
double* x; // current solution
double* r; // residuals
double* p; // direction
double* q; // support vector, q = A * p

// CG sequential kernel
enum ResumePoint{RESUME_FROM_START, RESUME_FROM_CGBODY, RESUME_FROM_CGEND};
ResumePoint resume_from;

// Instance variables are required to recover the previous state
unsigned ipm_current_iteration; // current IPM iteration
unsigned cg_current_iteration; // current CG iteration
double cg_alpha; // CG movement
double cg_rho; // CG direction

/**
 * Create the vectors,
 */
protected: virtual void initialise(Task_Initialisation* task);

/**
 * Implements the sequential part of the CG kernel
 */
protected: virtual void worker_cgkernel(Task_Multiplication* task_mult);

/**
 * Asks that the following operation must be a multiplication of the matrix
 */
private: Task_Multiplication* task_mult;
protected: void matrix_multiply(double* x, double* y);

public:
/**
 * Constructor
 * @param parallel current instance of the Parallel wrapper
 * @param cpu_id pin the worker to the specified cpu_id
 * @param status shared state with the emitter
 * @param ref_task_counter counter to distinct the numbers of active workers
 */
SuperWorker(Parallel& parallel, int cpu_id, TaskStatus& status, atomic_t& ref_task_counter);

/**
 * Destructor
 */
~SuperWorker();

};
Parallel::Parallel()
    // parse args
    num_workers = get_configuration().get_parallelism_grade();
    current_status = TASK_STATUS_INIT;
    current_task = NULL;
    atomic_set(&current_active_workers, 0);

    // init the farm
    farm = new ff_farm<Scheduler>();
    Scheduler& scheduler = *(Scheduler*) farm->getlb();
    scheduler.set_size_workers(num_workers);
    // scheduler.set_cgmast(1);

    // add the emitter
    emitter = new Emitter(0, num_workers, current_status,
                          current_active_workers, current_task, scheduler);
    farm->add_emitter(emitter);

    // create the workers
    std::vector<int> threads;
    cpu_topology::threadlist(threads, 0, cpu_topology::MODE_MAX_AVAIL);
    workers.push_back(
        new SuperWorker(*this, threads[0], current_status,
                        current_active_workers));
    for(int i = 1; i < num_workers; i++){
        workers.push_back(
            new Worker(threads[i], current_status, current_active_workers));
    }
    farm->add_workers(workers);

Parallel::~Parallel(){
    delete farm; farm = 0;
    delete emitter; emitter = 0;
    delete current_task; current_task = 0;
}

void Parallel::touch(){ }
if(farm->run_and_wait_end() < 0){
    throw std::runtime_error("[Parallel::execute] Error running the farm");
}

void Parallel::set_output(double zeta){ output_z = zeta; }

Emitter::Emitter(int cpu_id_, int num_workers_, TaskStatus& status_, atomic_t& completed_tasks_, Task*& task_ptr_, Scheduler& scheduler_):
    cpu_id(cpu_id_),
    num_workers(num_workers_),
    status(status_),
    waiting_for_tasks(completed_tasks_),
    task_args(task_ptr_),
    scheduler(scheduler_)
{ }

int Emitter::svc_init(){ ff_mapThreadToCpu(cpu_id); return 0;}

void* Emitter::svc(void* args){
    while(status!=TASK_STATUS_END){
        switch(status){
        case TASK_STATUS_INIT: {
            Task_Initialisation* task_init = static_cast<Task_Initialisation*>(
                task_args);
            atomic_set(&waiting_for_tasks, num_workers);
            int partition_size = (task_init->n / num_workers);
            int partition_size_last = partition_size + task_init->n %
                num_workers;
            for(int i = 0; i < num_workers; i++){
                Task_Initialisation* task = new Task_Initialisation(*task_init);
                task->first_column = i * partition_size;
                task->num_columns = (i < num_workers -1) ? partition_size :
                    partition_size_last;
                ff_send_out(task);
            }
            // wait for completion
            while(atomic_read(&waiting_for_tasks));
            // remove the makea wrapper
            MakeaWrapper::remove();
            // change the actual type of the task args
            delete task_init;
            task_args = new Task_Multiplication();
            // next operation will start the conjugate gradient
            status = TASK_STATUS_CGKERNEL;
            break;
        }
        
        break;
    }
case TASK_STATUS_MATRIX_MULT: {
    atomic_set(&waiting_for_tasks, num_workers);
    for(int i = 0; i < num_workers; i++){
        ff_send_out(task_args);
    }
    // wait for completion
    while(atomic_read(&waiting_for_tasks));
    // after a multiplication it always follows the sequential routine
    status = TASK_STATUS_CGKERNEL;
    break;
}

case TASK_STATUS_CGKERNEL:{
    atomic_set(&waiting_for_tasks, 1);
    scheduler.pick_cgmaster_next();
    ff_send_out(task_args);
    // wait for completion
    while(atomic_read(&waiting_for_tasks));
    break;
}

case TASK_STATUS_END:{
    // do nothing -> will terminate in the next iteration
    break;
}

default:
    throw std::runtime_error("[Emitter::svc] unrecognised status");
} // end switch
} // end if

delete task_args; task_args = 0;
return NULL; // stop the framework
}

Worker::Worker(int cpu_id_, TaskStatus& status_, atomic_t& ref_task_cntr_):
    cpu_id(cpu_id_), status(status_), waiting_for_tasks(ref_task_cntr_){
    local_y = NULL;
    local_N = 0;
    matrix_part = NULL;
}
Worker::~Worker(){
    delete matrix_part; matrix_part = 0;
    delete local_y; local_y = 0;
}
int Worker::svc_init()
{
    debug ( mutex.lock(); std::cout << "[worker::svc_init] Mapping worker id: " << get_my_id() << " to physical thread: " << cpu_id << std::endl; mutex.unlock(););
    ff_mapThreadToCpu(cpu_id);
    return 0;
}

void* Worker::svc(void* args){
    switch(status){
    case TASK_STATUS_INIT: {
        initialise(static_cast<Task_Initialisation*>(args));
        break;
    }
    case TASK_STATUS_MATRIX_MULT: {
        Task_Multiplication* task = static_cast<Task_Multiplication*>(args);
        worker_multiply(task->x, task->y);
        break;
    }
    case TASK_STATUS_CGKERNEL: {
        worker_cgkernel(static_cast<Task_Multiplication*>(args));
        break;
    }
    case TASK_STATUS_END: { /* throws an exception */
        // (...) 
        break;
    }
    default:
        throw std::runtime_error("[worker::svc] unrecognised task");
        break;
    }
}

void Worker::initialise(Task_Initialisation* task_init){
    matrix_part = new Matrix(
        task_init->n,
        task_init->nonzer,
        task_init->shift,
        task_init->rowblocks,
        task_init->colblocks,
        task_init->first_row,
        task_init->num_rows,
        task_init->first_column,
        task_init->num_columns
    );
    local_y = new double[local_N = task_init->n];
    delete task_init; task_init = 0;
}

void Worker::worker_multiply(double* x, double* restrict y){
    double* restrict y_local = this->local_y;
    matrix_part->multiply(x, y_local);
    mutex.lock();
    for(int i = 0; i < local_N; i++) y[i] += y_local[i];
    mutex.unlock();
void Worker::worker_cgkernel(Task_Multiplication*):
    /* throws an exception */
    // (...) }

/~

SuperWorker::SuperWorker(Parallel& parallel_, int cpu_id_, TaskStatus& status_,
    atomic_t& ref_task_cntr_):
    Worker(cpu_id_, status_, ref_task_cntr_),
    parallel(parallel_),
    N(parallel.input_N),
    niter(parallel.input_niter),
    lambda(parallel.input_lambda),
    cg_iterations(get_configuration().get<unsigned>("cgiterations"))
{
    // the initialisation() method will create the vectors
    b = x = r = p = q = NULL;

    // init cg instance vars
    resume_from = RESUME_FROM_START;
    ipm_current_iteration = 0;
    cg_current_iteration = 0; // current CG iteration
    cg_alpha = 0; // movement
    cg_rho = 0; // direction

    // multiplication params will be given later
    task_mult = NULL;

    SuperWorker::~SuperWorker(){
        delete[] b; b = NULL;
        delete[] x; x = NULL;
        delete[] r; r = NULL;
        delete[] p; p = NULL;
        delete[] q; q = NULL;
    }

    void SuperWorker::initialise(Task_Initialisation* task){
        Worker::initialise(task);

        if(b) { delete b; }
        if(x) { delete x; }
        if(r) { delete r; }
        if(p) { delete p; }
        if(q) { delete q; }

        // create the vectors
        b = new double[N];
        x = new double[N];
        r = new double[N];
        p = new double[N];
    }
```c++
q = new double[N];
resume_from = RESUME_FROM_START;
}

void SuperWorker::worker_cgkernel(Task_Multiplication* task_mult){
    this->task_mult = task_mult;
    // disable aliasing
    double* restrict b = this->b; // samples
    double* restrict x = this->x; // current solution
    double* restrict r = this->r; // residuals
    double* restrict p = this->p; // direction
    double* restrict q = this->q; // support vector, q = A * p

    switch(resume_from){
    case RESUME_FROM_START:{
        // restore b = [1 ... 1]'
        for(unsigned i = 0; i < N; i++){ b[i] = 1; }
        parallel.timer.start();
        // Start the conjugate gradient method
        cg_rho = 0; // direction
        memset(x, 0, N*sizeof(double)); // reset x = [0, ..., 0]
        // first iteration of the CG
        cg_current_iteration = 1;
        for(unsigned j = 0; j < N; j++){
            r[j] = p[j] = b[j];
            cg_rho += pow(b[j], 2);
        }
        resume_from = RESUME_FROM_CGBODY;
        matrix_multiply(p,q);
        break;
    }

    case RESUME_FROM_CGBODY:{
        double cg_pAp = 0; // p' * q = p'A q
        for(unsigned j = 0; j < N; j++){ cg_pAp += p[j] * q[j]; }
        cg_alpha = cg_rho / cg_pAp; // = rho / p'q = rho / p'Ap
        // next solution
        for(unsigned j = 0; j < N; j++){ x[j] += cg_alpha * p[j]; }
        cg_current_iteration++;
        if(cg_current_iteration <= cg_iterations){ // next iteration
            double rho0 = cg_rho;
            // r = r - alpha * q
            for(unsigned j = 0; j < N; j++){ r[j] = r[j] - cg_alpha * q[j]; }
            // rho = r' * r
            cg_rho = 0;
            for(unsigned j = 0; j < N; j++){ cg_rho += pow(r[j], 2); }
            double beta = cg_rho / rho0;
        }
    }
```
for(unsigned j = 0; j < N; j++) { p[j] = r[j] + beta * p[j]; }

// redundant, clarifies it has to repeat a new CG iteration
resume_from = RESUME_FROM_CGBODY;
matrix_multiply(p, q);
} else { // exit from the CG iterations
// q = A x
resume_from = RESUME_FROM_CGEND;
matrix_multiply(x, q);
}
break;
}

break;
}
case RESUME_FROM_CGEND:
  // r = b - A x
for(unsigned j = 0; j < N; j++) { r[j] = b[j] - q[j]; }

  // rnorm = || b - A x ||
  double rnorm = 0;
  for(unsigned j = 0; j < N; j++) { rnorm += pow(r[j], 2); }
  rnorm = sqrt(rnorm);

  // return rnorm;

  // --- Continue with the inverse power method ---
  // compute zeta = shift + 1/(b*x)
  double bx = 0;
  for(unsigned j = 0; j < N; j++) { bx += b[j] * x[j]; }
  double zeta = lambda + 1 / bx;

  // print the current iteration, ||r|| and zeta
  if(ipm_current_iteration > 0)
    printf("it: %d, ||r||: %E, zeta: %.15f
", ipm_current_iteration, rnorm, zeta);

  // compute the norm2 of x, ||x||
  double xnorm = 0;
  for(unsigned j = 0; j < N; j++) { xnorm += pow(x[j], 2); }
xnorm = sqrt(xnorm);

  // update b = x / ||x||
  for(unsigned j = 0; j < N; j++) { b[j] = x[j] / xnorm; }

  ipm_current_iteration++;
  if(ipm_current_iteration <= niter) { // next IPM iteration
    // Start the conjugate gradient method
    cg_rho = 0; // direction
    memset(x, 0, N*sizeof(double)); // reset x = [0, ..., 0]

    // first iteration of the CG
    cg_current_iteration = 1;

    for(unsigned j = 0; j < N; j++) {
      r[j] = p[j] = b[j];
      cg_rho += pow(b[j], 2);
    }

    resume_from = RESUME_FROM_CGBODY;
410        matrix_multiply(p, q);
411    } else { // the end
412        parallel.set_output(zeta);
413        parallel.timer.stop();
414        status = TASK_STATUS_END; // ends the farm
415    }
416    break;
417  }
418  
419  
420  
421  
422  
423 void SuperWorker::matrix_multiply(double* x, double* y){
424    memset(y, 0, N * sizeof(double));
425    this->task_mult->x = x;
426    this->task_mult->y = y;
427    status = TASK_STATUS_MATRIX_MULT;
428  }
Implementation of the sequential algorithm for the MultiGrid (MG) kernel.

```cpp
class Sequential : public my::mg::Base{
protected:
    /*
    * The number of levels for this system, i.e. log2(n).
    */
    const std::size_t mg_levels;

    /*
    * A general memory portion, employed by the grid to save temporary values.
    * Since this algorithm sequentially performs each operation, grids access
    * the buffer in exclusive manner. The size of this area is equivalent to
    * five rows of the biggest grid.
    */
    double* shared_buffer;

    /*
    * Array of the residuals.
    * This array contains log2(n) pointers to Grids, representing the residual
    * of each level. Residual for the last level represents the error between
    * the current solution and the vector of the constant terms: r[n−1] = b −
    * Ax. Coarser residuals are obtained through the projection (restriction)
    * operator on the immediately finer level, reducing the size of each grid
    * of 1/8 (1/2 for each dimension).
    * The logical size of the residual grid at the i−th level will be:
    * 2^(i+1) x 2^(i+1) x 2^(i+1).
    */
    SequentialGrid3D** residual;

    /*
    * Array of the corrections.
    * This array contains log2(n) pointers to Grids, representing the correction
    * on the solution of each level. For the finest level, the grid is
    * the current solution of the system Ax − b. In coarser levels, the grids
    * represent the correction on the corresponding residual, obtained through
    * the recursive solving of the system and one iteration of the iterative
    * method (smoothing).
    * The logical size of the correction grid for the i−th level will be
    * 2^(i+1) x 2^(i+1) x 2^(i+1).
    */
    SequentialGrid3D** correction;

    /*
    * Performs the sequential MG algorithm
    */
    virtual void execute();

    /*
    * Resets all grids and loads involved elements in memory.
    */
};
```
virtual void touch();

/**
  * Internal timers
  */
  // (...)

public:
/**
  * Default constructor
  */
  Sequential();

/**
  * Default destructor
  */
  virtual ~Sequential();
};

Listing A.11: MG / sequential.cpp

Sequential::Sequential() : Base(), mg_levels(log2(mg_size)) {
  // shared buffer to contain temporary values
  shared_buffer = SequentialGrid3D::buffer_alloc(mg_size);

  // init the involved grids
  residual = new SequentialGrid3D*[mg_levels]; // residual
  correction = new SequentialGrid3D*[mg_levels]; // solution / corrections
  size_t current_level = mg_size;
  for(int i = mg_levels -1; i >= 0; i--){
    residual[i] = new SequentialGrid3D(
      current_level,
      coeff_residual,
      coeff_smoothing,
      shared_buffer
    );
    correction[i] = new SequentialGrid3D(
      current_level,
      coeff_residual,
      coeff_smoothing,
      shared_buffer
    );
    current_level = current_level / 2;
  }

  // init internal timers
  // (...)
}

Sequential::~Sequential() {
  // remove allocated grids
  for(size_t i = 0; i < mg_levels; i++){
    delete residual[i]; residual[i]=0;
    delete correction[i]; correction[i]=0;
  }
  delete[] residual;
  delete[] correction;
  //delete [] shared_buffer;
void Sequential::execute(){
    SequentialGrid3D& solution = *(correction[mg_levels -1]); // solution
    SequentialGrid3D& error = *(residual[mg_levels -1]); // error
    timer.start();
    // r = v, u is zero
    error.set(constant_terms, constant_terms_sz);
    error.update_boundaries();
    for(size_t it = 1; it <= mg_iterations; it ++){
        if(it == 1 || it % 5 == 0){ cout << "Iteration:␣" << it << endl; }
        for(size_t i = mg_levels -1; i > 0; i--){ // down cycle, projection
            residual[i-1]->restriction(*residual[i]);
        }
        // apply the correction in the coarsest grid
        correction[0]->smoother(*residual[0]);
        for(size_t i = 1; i < mg_levels; i++){ // upper cycle
            if(i < mg_levels -1){
                correction[i]->interpolation(*correction[i-1]); // prolongate the coarser solution
                residual[i]->subtract_residual(*correction[i]);
            } else { // last iteration
                correction[i]->add_interpolation(*correction[i-1]);
                residual[i]->residual(*correction[i], constant_terms,
                constant_terms_sz);
            }
        }
        // compute the next solution z = z + Sr
        correction[i]->add_smoother(*residual[i]);
    }
    // evaluate the error for the current solution
    error.residual(solution, constant_terms, constant_terms_sz);
    set_computed_norm_l2(error.normL2());
    timer.stop();
}
void Sequential::touch(){
    // resets the grids
    for(size_t i = 0; i < mg_levels; i++){
        residual[i]->erase();
        correction[i]->erase();
    }
    // resets the timers
    // (...)
Parallel grid

For brevity reasons, it is only reported the Grid 3D employed in the parallel algorithm. It is similar to the data structure of the sequential algorithm. The main difference is the parallel algorithm may work on arbitrary partitions and not in the whole grid of a certain level. It additionally considers misalignments among the grids of different levels and the boundary exchange among neighbours of the same level. The sequential grid is more specialised, as there are less cases to treat.

Listing A.12: MG / parallel_grid.h

```c
/**
 * Parallel Grid — October 28th, 2012
 *
 * This is the parallel version of the grid data structure. The multigrid parallel algorithm employs this data structure to perform the operations among the involved 3d grids of values: interpolation, smoothing, restriction and residual. See the original NPB specification for their meaning.

 * The parallel grid may represent a single partition of the whole 3D grid. Each partition may contain one or more depths of the logical grid. As each partition replicates the data of the boundary values, the whole set of partitions composing a logical grid should be linked between consecutive neighbours. The aim is to exchange the boundary depths, required by the involved operations.

 * Operations such as the residual and the smoothing must operate on compatible partitions of the same size and the same covered interval. To avoid additional overheads, there are no extra checks if the intervals are compatible. At the end of the computation, the boundary values are automatically sent to its neighbours. You need to manually invoke the method exchange_boundaries() for each touched partition of dimension N.

 * The interpolation must operate on partitions of dimension N/2. The input involved intervals are correctly checked and errors due to misaligned values cannot occur. Be sure to pass non overlapping partitions to the method add_interpolation, otherwise some intervals may be added multiple times. Note that, at the end of the computation, the boundary depths are not automatically sent to its neighbours. You need to manually invoke the method exchange_boundaries() for each touched partition of dimension N.

 * The same considerations that are valid for the interpolation, holds for the restriction operator. In this case, the input partition must be of dimension 2*N. Input partitions may be not a subset, be not aligned or even relate to a compatible interval. The involved intervals are always checked and correctly computed. As for the interpolation, the operator does not automatically update the boundary values at the end of the computation. Invoke the method update_local_boundaries() to do so.

 * This class is not thread-safe. However, if each thread works on a different partition of the whole logical grid, then protection and synchronisation methods are not required. Indeed, each partition can touch only the local values and the boundary depths of its neighbours. This scenario applies for the implemented multigrid algorithm.

 * The class provides several static methods to sequentially perform the offered operations on groups of partitions.
 */
```
A single point in the 3D Cartesian space is given by its axis coordinates and its value.

```cpp
struct SparsePoint{
    int i, j, k;
    double value;
};
```

```cpp
class ParallelGrid3D {
private:
    // lengths
    const std::size_t N;
    const std::size_t distance_row;
    const std::size_t distance_depth;
    const std::size_t depth_start;
    const std::size_t depth_length;

    // values and temporary buffers to precompute the result
    double* values;
    double* buffers;

    // neighbours
    ParallelGrid3D* neighbour_previous;
    double* neighbour_ptr_previous;
    ParallelGrid3D* neighbour_next;
    double* neighbour_ptr_next;

    // coefficients of the smoother and the residual operator
    const double* coeff_residual;
    const double* coeff_smoothing;

    // flag to determine if the current buffer is private
    const bool shared_buffer;

    /**
     * Updates both local boundary rows and columns for the body values, i.e.
     * starting from depth +1 to depth + length +1. The boundary depths are not
     * touched, they should be updated by the neighbour nodes through the
     * method exchange_boundaries().
     */
    void update_local_boundary_body();

    /**
     * Prints onto the stdout all the values of this grid
     * @param internal_from: first depth to print, a valid value starts from 0.
     * @param internal_length: amount of depths to print
     */
    void print_local_values(std::size_t internal_from, std::size_t internal_length) const;

    /**
     * Computes the bounds for the interpolation operator
     */
    static void interpolation_bounds(ParallelGrid3D* out, ParallelGrid3D* in,
                                       std::size_t* out_start, std::size_t* in_start, std::size_t* in_length,
                                       bool* skip_first, bool* skip_last);
```
```cpp
/**
 * Computes the bounds for the restriction operator
 */
static void restriction_bounds(ParallelGrid3D* out, ParallelGrid3D* in, std::size_t* out_start, std::size_t* out_length, std::size_t* in_start, bool* skip_first, bool* skip_last);

/**
 * Unchecked get and set, with internal depths
 * @param i: row index in [0, N+2]
 * @param j: column index in [0, N+2]
 * @param k: depth index in [0, depth_length +2]
 */
inline double get0(int i, int j, int ik) const{
    return values[ik*distance_depth + i * distance_row + j];
}
inline void incr0(int i, int j, int ik, double v){
    values[ik*distance_depth + i * distance_row + j] += v;
}
inline void set0(int i, int j, int ik, double v){
    values[ik*distance_depth + i * distance_row + j] = v;
}

public:

/**
 * Constructor
 * @param N: dimension of the logical grid
 * @param coeff_residual: array of the four coefficients for the residual operator
 * @param coeff_smoothing: array of the four coefficients for the smoothing operator
 * @param buffer: a proper buffer to use as working set, must be allocated in advance with the static method buffer.alloc()
 * @param start: first depth covered in this partition. Note that depths start from 1.
 * @param length: length of this partition, i.e. the number of depths composing this partition.
 */
ParallelGrid3D(std::size_t N, const double* coeff_residual, const double* coeff_smoothing, double* buffer = NULL, std::size_t start =1, std::size_t length =0);

/**
 * Destructor
 */
virtual ~ParallelGrid3D();

/**
 * Adds the results of the interpolation on the given grid for the values inside the bounds.
 * Note: this method does not automatically exchange the boundaries with its neighbours at the end of the computation. To achieve that, manually invoke the method exchange_boundaries();
 * @param u: the input grid. Its dimension must be (N/2)^3
 */
void add_interpolation(ParallelGrid3D& u);
```
```cpp
void add_interpolation(ParallelGrid3D* u[], std::size_t u_sz){
    ParallelGrid3D* out[] = {this};
    add_interpolation(out, 1, u, u_sz);
}

void interpolation(ParallelGrid3D& u);
```

void restriction(ParallelGrid3D* u[], std::size_t u_sz){
    ParallelGrid3D* out[] = {this};
    restriction(out, 1, u, u_sz);
}

void smoother(ParallelGrid3D& u);

void subtract_residual(ParallelGrid3D& u);

void add(SparsePoint* v, std::size_t v_sz);
void set(SparsePoint* v, std::size_t v_sz);

void exchange_boundaries();

inline void update_local_boundaries(){
    update_local_boundary_body(); exchange_boundaries();
}

* Projects a group of grids into this grid and updates the boundaries at
  the end of the computation.
* @param u: array of non overlapping grids of dimension 2*N
* @param u_sz: size of the array u
*/

void restriction(ParallelGrid3D* u[], std::size_t u_sz)
{    ParallelGrid3D* out[] = {this};
    restriction(out, 1, u, u_sz);
}

/**
 * Applies the smoothing operator to the grid u and saves the results in
 * this grid
 * @param u: the input grid
 */
void smoother(ParallelGrid3D& u);

/**
 * Subtracts to this grid the residual R u, where R is the residual
 * operator.
 * @param u: grid pointing to the current solution/correction
 */
void subtract_residual(ParallelGrid3D& u);

/**
 * Directly accesses and sets the elements
 * @param i: row index in [0,N+2]
 * @param j: column index in [0, N+2]
 * @param k: depth index in [depth_start, depth_start + depth_length +2]
 */
double get(int i, int j, int k) const;
void set(int i, int j, int k, double v);

/**
 * Attempts to increment / set a group of values, as specified by the
 * SparsePoint array. These methods do not automatically update the bounds.
 * Values out of the bounds of this grid are ignored.
 * @param v: array of sparse points
 * @param v_sz: the size of the array
 */
void add(SparsePoint* v, std::size_t v_sz);
void set(SparsePoint* v, std::size_t v_sz);

/**
 * Sends the upper and below boundaries to its neighbours. If there are no
 * neighbours, then it will exchange the boundaries locally.
 */
void exchange_boundaries();

/**
 * Updates the local boundaries of this grid and sends the upper/below
 * partitions to its neighbours.
 */
inline void update_local_boundaries()
{
    update_local_boundary_body(); exchange_boundaries();
}
*/ Prints onto the stdout all the local values of this grid
*/

void print_local_values() const{
    print_local_values(0, depth_length+2);
}

/**
* Computes the local norm 2.
*/

double local_normL2(){
    ParallelGrid3D* a[] = {this};
    return normL2(a,1);
}

/**
* Sets the neighbour nodes. It will exchange the boundary values during
* the computation.
* @param previous: the grid which owns and requires the upper depths
* @param next: the grid which owns and requires the below depths
*/

void set_neighbours(ParallelGrid3D* previous, ParallelGrid3D* next);

/**
* Sets all values, including the boundaries, to zero.
*/

void erase();
/*
 * Sequentially applies the residual operator to the given set of grids
 * @param r : array of residual grids
 * @param u : array of the solution grids
 * @param sz : size of the array r and u. The two arrays must have the same
 * size.
 * @param v : array of the sparse fixed terms
 * @param v_sz : size of the array v
 */
static void residual(ParallelGrid3D* r[], ParallelGrid3D* u[], std::size_t sz, SparsePoint v[], std::size_t v_sz);

/*
 * Sequentially applies the smoothing operator to the specified set of
 * grids, i.e. u_next = S(u_prev), where S is the smoothing operator.
 * @param u_next : the grids where computed values will be stored
 * @param u_prev : the input grids
 * @param sz : size of both arrays r and u.
 */
static void smoother(ParallelGrid3D* u_next[], ParallelGrid3D* u_prev[], std::size_t sz);

/*
 * Sequentially performs the subtraction of the residual to the specified
 * set of grids, i.e. r -= R(u), where R is the residual operator.
 * @param r : array of the residual grids
 * @param u : array of the solution/correction grids
 * @param sz : size of both arrays r and u.
 */
static void subtract_residual(ParallelGrid3D* r[], ParallelGrid3D* u[], std::size_t sz);

/*
 * Attempts to set the given value in the corresponding grid, contained in
 * the array
 */
static void set(ParallelGrid3D* grids[], std::size_t grids_sz, int i, int j , int k, double v);

/*
 * Attempts to assign the given set of values to the corresponding grids.
 * This method does not automatically updates the boundaries.
 */
static void set(ParallelGrid3D* grids[], std::size_t grids_sz, SparsePoint v[], std::size_t v_sz);
static void set_circular_neighbours(ParallelGrid3D* grids[], std::size_t grids_sz);

/**
 * Sequentially computes the norm L2 for the given set of grids
 */
static double normL2(ParallelGrid3D* grids[], std::size_t grids_sz);

/**
 * Sequentially updates the boundaries for the given set of grids
 */
static void update_boundaries(ParallelGrid3D* grids[], std::size_t grids_sz);

/**
 * Prints onto the stdout all the values for the array of given grids
 */
static void print_values(ParallelGrid3D* grids[], std::size_t grids_sz);

/* BUFFER ALLOCATION */

static double* buffer_alloc(std::size_t N);

static void buffer_free(double* buffer);

};

Listing A.13: MG / parallel_grid.cpp

static const std::size_t CACHE_LINE = opt_cache_level1_linesize;
static const std::size_t doubles_per_block = CACHE_LINE / sizeof(double);

using std::cout;
using std::endl;
using std::setprecision;

using std::size_t;

static const double COEFF_INTERP_C0 = 1.;
static const double COEFF_INTERP_C1 = 1./2;
static const double COEFF_INTERP_C2 = 1./4;
static const double COEFF_INTERP_C3 = 1./8;

static const double COEFF_PROJECT_C0 = 1./2;
static const double COEFF_PROJECT_C1 = 1./4;
static const double COEFF_PROJECT_C2 = 1./8;
static const double COEFF_PROJECT_C3 = 1./16;

/* ***********************************************
 * SUPPORT FUNCTIONS
 * ***********************************************
namespace {
    // support functions

    // convert an object\integer in a string
    std::string str(const T& value){
        std::stringstream stream;
        stream << value;
        return stream.str();
    }

    // [add_interpolation] compute the even depths
    inline void _add_interp_compute_even_depth(
        double* restrict row_thist,
        double* restrict row_input,
        double* restrict buffer,
        const size_t input_N,
        const std::size_t out_distance_row,
        const double c0, const double c1, const double c2)
    {
        (void) CACHE_ALIGNED(row_this);
        (void) CACHE_ALIGNED(row_input);
        (void) CACHE_ALIGNED(buffer);

        for(size_t j = 0, J = 0; j < input_N+1; j++, J+=2){
            row_this[J] += c0 * row_input[j];
            row_this[J+1] += c1 * (row_input[j] + row_input[j+1]);
        }

        // odd rows, even depth
        CACHE_ALIGNED(row_this_current += out_distance_row);
        for(size_t j = 0, J = 0; j < input_N+1; j++, J+=2){
            row_this[J] += c1 * buffer[j];
            row_this[J+1] += c2 * (buffer[j] + buffer[j+1]);
        }
    }

    // [add_interpolation] compute the odd depths
    inline void _add_interp_compute_odd_depth(
        double* restrict row_this,
        double* restrict row_input,
        double* restrict buffer,
        const size_t input_N,
        const size_t out_distance_row,
        const size_t input_distance_depth,
        const double c1, const double c2, const double c3)
    {
        (void) CACHE_ALIGNED(row_this);
        (void) CACHE_ALIGNED(row_input);
        (void) CACHE_ALIGNED(buffer);
// even rows, odd depth
for(size_t j = 0, J = 0; j < input_N+1; j++, J+=2){
    row_this[J] += c1 * (row_input[j] +
                    row_input[j + input_distance_depth]);
    row_this[J+1] += c2 * (row_input[j] +
                        row_input[j+1] +
                        row_input_current[j+1 + input_distance_depth]);
}

// odd rows, odd depth
CACHE_ALIGNED(row_this += out_distance_row);
for(size_t j = 0, J = 0; j < input_N+1; j++, J+=2){
    row_this[J] += c2 * buffer[j];
    row_this[J+1] += c3 * (buffer[j] + buffer[j+1]);
}

// [interpolation] precompute c1, c2 values
inline void _interp_compute_buffer(
    double* restrict buffer_this,
    double* restrict buffer_next,
    double* restrict row_input,
    const size_t input_N,
    const std::size_t input_distance_row,
    const std::size_t input_distance_depth)
{
    (void)CACHE_ALIGNED(buffer_this);
    (void)CACHE_ALIGNED(buffer_next);
    (void)CACHE_ALIGNED(row_input);

    // pre compute mixed values
    for(size_t j = 0; j < input_N+2; j++) // j refers to the input
        buffer_next[j] = buffer_this[j] + row_input[j+input_distance_row] +
                          row_input[j+input_distance_depth+input_distance_row];
}

// [interpolation] compute the even depths
inline void _interp_compute_even_depth(...){
    // Similar to _add_interp_compute_even_depth, replace the additions with
    // the assignments (...)
}

// [interpolation] compute the odd depths
inline void _interp_compute_odd_depth(...){
    // Similar to _add_interp_compute_odd_depth, replace the additions with
    // the assignments (...)
}

} // end (support functions)
// public constructor
ParallelGrid3D::ParallelGrid3D(size_t N_, const double* cf_residual_, const
double* cf_smoothing_, double* bf_, size_t depth_start_, size_t
depth_length_) :
    N(N_),
    distance_row(ceil(((double) N+2) / doubles_per_block) * doubles_per_block),
depth_row(distance_row * (N+2)),
depth_length(depth_length_? depth_length_ : N),
    neighbour_previous(0),
    neighbour_ptr_previous(0),
    neighbour_next(0),
    neighbour_ptr_next(0),
    coeff_residual(cf_residual_),
    coeff_smoothing(cf_smoothing_),
    shared_buffer(!bf_)
{
    // Constructor body
    if(depth_start_ < 1) {/* throws an exception (...) */ }
    size_t mem_size_per_row = distance_row * sizeof(double);
    int result = posix_memalign((void**) &values, CACHE_LINE, mem_size_total);
    if(result!=0) {/* throws an exception (...) */ }
    memset(values, 0, memory_size_total);
    if(!shared_buffer) buffers = buffer_alloc(N+1);
}

ParallelGrid3D::~ParallelGrid3D() {
} free(values); values = 0;
if(!shared_buffer) {buffer_free(buffers);} }

void ParallelGrid3D::add_interpolation(ParallelGrid3D& input){
    // check the partition bounds
    size_t input_offset(0), this_offset(0), length(0);
    bool skip_first(0), skip_last(0);
    interpolation_bounds(this, &input, &this_offset, &input_offset, &length,
                          &skip_first, &skip_last);
    if(!length) return; /* out of the grid interval */

    // coefficients
    const double c0 = COEFF_INTERP_C0;
    const double c1 = COEFF_INTERP_C1;
    const double c2 = COEFF_INTERP_C2;
    const double c3 = COEFF_INTERP_C3;

    // buffer for temps
    double* restrict buffer_this_depth = CACHE_ALIGNED(buffers);
    double* restrict buffer_next_depth =
        CACHE_ALIGNED(buffers + input.distance_row);

    // pointer to current depth in this grid
    double* restrict depth_this_current =
        values + (this_offset*distance_depth);
    // pointer to current depth in the input grid
    double* restrict depth_input_current =

input.values + (input_offset*input.distance_depth);

{ // first iteration

if(!skip_first){ // do not skip the first depth
// move by rows
  double* restrict row_this = CACHEAligned( depth_this_current );
  double* restrict row_input_current = CACHEAligned( depth_input_current );
  for(size_t i = 0; i < input.N+1; i ++){
    // pre-compute mixed values
    _interp_compute_buffer(buffer_this_depth, buffer_next_depth,
                         row_input_current, input.N, input.distance_row,
                         input.distance_depth);

    // even depth
    _add_interp_compute_even_depth(row_this, row_input_current,
                                    buffer_this_depth, input.N, distance_row, c0, c1, c2);

    // odd depth
    if(!skip_last || length != 1){ // skip?
      _add_interp_compute_odd_depth(row_this + distance_depth, row_input_current, buffer_next_depth, input.N,
                                     distance_row, input.distance_depth, c1,c2,c3);
    }

    // next row
    row_this += 2*distance_row;
    row_input_current += input.distance_row;
  }

// move ahead to the next depth
  depth_this_current += 2*distance_depth;
  depth_input_current += input.distance_depth;
} else { // skip the first depth because its starting depth is even
  // move by rows
  double* restrict row_this = CACHEAligned( depth_this_current );
  double* restrict row_input_current = CACHEAligned( depth_input_current );
  for(size_t i = 0; i < input.N+1; i ++){
    // pre-compute mixed values
    for(size_t j = 0; j < input.N+2; j++) { // j refers to the input
      buffer_next_depth[j] = row_input_current[j] +
                             row_input_current[j+input.distance_row] +
                             row_input_current[j+input.distance_depth] +
                             row_input_current[j+input.distance_depth+input.distance_row];
    }

    // it cannot occur that skip_last == true, as this is the first
    // iteration and length > 0
    _add_interp_compute_odd_depth(row_this, row_input_current, buffer_next_depth, input.N, distance_row,
                                   input.distance_depth, c1,c2,c3);
```c
    // next row
    row_this += 2*distance_row;
    row_input_current += input.distance_row;
    
    // move ahead to the next depth
    depth_this_current += distance_depth; // distance is 1, not 2
    depth_input_current += input.distance_depth;

} // end if (skip_first)

} // end of the first iteration

// body iterations, blocks are aligned
for(size_t k = 1; k < length -1; k++){

    // move by rows
    double* restrict row_this = CACHE_ALIGNED( depth_this_current );
    double* restrict row_input_current = CACHE_ALIGNED( depth_input_current );
    for(size_t i = 0; i < input.N+1; i++){

        // pre compute mixed values
        _interp_compute_buffer(buffer_this_depth, buffer_next_depth,
            row_input_current, input.N, input.distance_row,
            input.distance_depth);

        // even depths
        _add_interp_compute_even_depth(row_this, row_input_current,
            buffer_this_depth, input.N, distance_row, c0, c1, c2);

        // odd depths
        _add_interp_compute_odd_depth(row_this + distance_depth,
            row_input_current, buffer_next_depth, input.N,
            distance_row, input.distance_depth, c1, c2, c3);

        // next row
        row_this += 2*distance_row;
        row_input_current += input.distance_row;
    }

    // move ahead to the next depth
    depth_this_current += 2*distance_depth;
    depth_input_current += input.distance_depth;

} // end of the body iterations

if(length>1){ // last iteration

    // move by rows
    double* restrict row_this = CACHE_ALIGNED( depth_this_current );
    double* restrict row_input_current = CACHE_ALIGNED( depth_input_current );
    for(size_t i = 0; i < input.N+1; i++){

        // pre compute mixed values
        _interp_compute_buffer(buffer_this_depth, buffer_next_depth,
            row_input_current, input.N, input.distance_row,
            input.distance_depth);

    } // end for (i)

    // even depths
    _add_interp_compute_even_depth(row_this, row_input_current,
        buffer_this_depth, input.N, distance_row, c0, c1, c2);

    // odd depths
    _add_interp_compute_odd_depth(row_this + distance_depth,
        row_input_current, buffer_next_depth, input.N,
        distance_row, input.distance_depth, c1, c2, c3);

    // next row
    row_this += 2*distance_row;
    row_input_current += input.distance_row;

    // move ahead to the next depth
    depth_this_current += 2*distance_depth;
    depth_input_current += input.distance_depth;

} // end if (length>1)
```
```c
    _interp_compute_buffer(buffer_this_depth, buffer_next_depth,
                      row_input_current, input.N, input.distance_row,
                      input.distance_depth);

    // even depths
    _add_interp_compute_even_depth(row_this, row_input_current,
                                   buffer_this_depth, input.N, distance_row, c0, c1, c2);

    // odd depths
    if(!skip_last){
        _add_interp_compute_odd_depth(row_this + distance_depth,
                                      row_input_current, buffer_next_depth, input.N,
                                      distance_row, input.distance_depth, c1, c2, c3);
    }

    // next row
    row_this += 2*distance_row;
    row_input_current += input.distance_row;
}

// end of the last iteration

void ParallelGrid3D::add_smother(ParallelGrid3D& input){
    // coefficients
    const double c0 = coeff_smoothing[0];
    const double c1 = coeff_smoothing[1];
    const double c2 = coeff_smoothing[2];
    // c3 is zero in the specification, but the program loads coeffs at runtime. The multiplication by zero would be performed if not
    // statically disabled.
    #ifdef MG_SMOOTHING_ENABLE_C3
        const double c3 = coeff_smoothing[3];
    #endif

    // buffer for temps
    double* restrict buffer_c1 = CACHE_ALIGNED(buffers);
    double* restrict buffer_c2 = CACHE_ALIGNED(buffers + distance_row);

    // pointer to current depth in the src grid
    double* depth_this_current = values + distance_depth;
    // pointer to current depth in the dest grid
    double* depth_input_current = input.values + distance_depth;

    // move by depth
    for(size_t k = 0; k < depth_length; k++){
        // move by rows
        double* restrict row_this_current =
            CACHE_ALIGNED( depth_this_current + distance_row );
        double* restrict row_input_current =
            CACHE_ALIGNED( depth_input_current + distance_row );

        for(size_t i = 0; i < N; i ++){
            // precompute c1 and c2 values
            for(size_t j = 0; j < N+2; j ++){
                buffer_c1[j] = row_input_current[j -distance_depth] +
                              row_input_current[j -distance_row] +
                              row_input_current[j +distance_row] +
                              row_input_current[j +distance_depth];
            }
            // compute c1 and c2
            row_input_current =
                buffer_c1 +
                c0 * row_input_current +
                c1 * depth_this_current +
                c2 * depth_input_current +
                c3 * depth_this_current * depth_input_current;
        }
    }
```
for(size_t j = 0; j < N+2; j ++){
    buffer_c2[j] =
        row_input_current[j -distance_depth -distance_row] +
        row_input_current[j -distance_depth +distance_row] +
        row_input_current[j +distance_depth -distance_row] +
        row_input_current[j +distance_depth +distance_row];
}

// smoothing
for(size_t j = 1; j < N+1; j++){
    row_this_current[j] += c0 * row_input_current[j] +
        c1 * (row_input_current[j-1] + row_input_current[j+1] +
              buffer_c1[j] ) +
        c2 * (buffer_c1[j-1] + buffer_c1[j+1] + buffer_c2[j]);
    #ifdef MG_SMOOTHING_ENABLE_C3
    row_this_current[j] += c3 * (buffer_c2[j-1] + buffer_c2[j+1]);
    #endif
} // end for (smoothing)

// next row
row_this_current += distance_row;
row_input_current += distance_row;
}

// move ahead to the next depth
depth_this_current += distance_depth;
depth_input_current += distance_depth;
}

update_local_boundaries();

void ParallelGrid3D::interpolation(ParallelGrid3D& input){
    // similar to the method add_interpolation, replace additions with
    // assignments ( ... )
}

void ParallelGrid3D::restriction(ParallelGrid3D& input){
    // check the partition bounds
    size_t input_offset(0), this_offset(0), length(0);
    bool skip_first(0), skip_last(0);
    restriction_bounds(this, &input, &this_offset, &length, &input_offset,
                       &skip_first, &skip_last);
    if(!length) return; // out of the grid interval
    // if (skip_last) length -= 1;

    // coefficients
    const double c0 = COEFF_PROJECT_C0;
    const double c1 = COEFF_PROJECT_C1;
    const double c2 = COEFF_PROJECT_C2;
    const double c3 = COEFF_PROJECT_C3;

    // buffer for temps
    double* restrict buffer_c1 = CACHE_ALIGNED(buffers);
    double* restrict buffer_c2 = CACHE_ALIGNED(buffers + input.distance_row);

    // pointer to current depth in this grid
    double* depth_this_current = values + this_offset*distance_depth;
    // pointer to current depth in the input grid
double* depth_input_current =
    input.values + input_offset*input.distance_depth;

if(skip_first){ // input and output partitions are misaligned
    double* restrict row_this_current =
        CACHE_ALIGNED( depth_this_current + distance_row );
    double* restrict row_input_current =
        CACHE_ALIGNED( depth_input_current + input.distance_row *2 );

    for(size_t i = 0; i < N; i ++){
        // precompute c1 and c2 values
        for(size_t j = 0; j < input.N+2; j ++){
            buffer_c1[j] = row_input_current[j-input.distance_row] +
                row_input_current[j+input.distance_row] +
                row_input_current[j+input.distance_depth];
        }
        for(size_t j = 0; j < input.N+2; j ++){
            buffer_c2[j] =
                row_input_current[j+input.distance_depth -input.distance_row] +
                row_input_current[j+input.distance_depth +input.distance_row];
        }

        // restriction
        for(size_t j = 1, J = 2; j < N+1; j=j_this++, J=j_input+=2){
            // do not change the +=, as it adds to the restriction
            // of skip_last
            row_this_current[j] += c0 * row_input_current[J] +
                c1 * (row_input_current[J-1] + row_input_current[J+1] +
                    buffer_c1[J] ) +
                c2 * (buffer_c1[J -1] +buffer_c1[J +1] +buffer_c2[J]) +
                c3 * (buffer_c2[J -1] + buffer_c2[J +1]);
        } // end for (restriction)

        // next row
        row_this_current += distance_row;
        row_input_current += input.distance_row*2;
    }

    // move ahead to the next depth
    depth_this_current += distance_depth;
    depth_input_current += input.distance_depth*2;

    // perform one iteration less
    length--;
}

} // end of first iteration

// handle the last depth separately
if(length==0) return; // avoids overflow
if(skip_last){ length--; }

// body
for(size_t k = 0; k < length; k++){
    // move by rows
    double* restrict row_this_current =
        CACHE_ALIGNED( depth_this_current + distance_row );
    double* restrict row_input_current =
for(size_t i = 0; i < N; i ++){
    
    // precompute c1 and c2 values
    for(size_t j = 0; j < input.N+2; j ++){
        buffer_c1[j] = row_input_current[j-input.distance_depth] +
        row_input_current[j-input.distance_row] +
        row_input_current[j+input.distance_row] +
        row_input_current[j+input.distance_depth];
    }
    for(size_t j = 0; j < input.N+2; j ++){
        buffer_c2[j] =
        row_input_current[j-input.distance_depth-input.distance_row] +
        row_input_current[j-input.distance_depth+input.distance_row] +
        row_input_current[j+input.distance_depth-input.distance_row] +
        row_input_current[j+input.distance_depth+input.distance_row];
    }

    // restriction
    for(size_t j = 1, J = 2; j < N+1; j++, J+=2){
        row_this_current[j] =
        c0 * row_input_current[J] +
        c1 * (row_input_current[J-1] + row_input_current[J+1] +
            buffer_c1[J]) +
        c2 * (buffer_c1[J -1] +buffer_c1[J +1] +buffer_c2[J]) +
        c3 * (buffer_c2[J -1] + buffer_c2[J +1]);
    } // end for (restriction)

    // next row
    row_this_current += distance_row;
    row_input_current += input.distance_row*2;
}

// move ahead to the next depth
depth_this_current += distance_depth;
depth_input_current += input.distance_depth*2;
}

if(skip_last){

    // move by rows
    double* restrict row_this_current =
      CACHE_ALIGNED( depth_this_current + distance_row );
    double* restrict row_input_current =
      CACHE_ALIGNED( depth_input_current + input.distance_row *2 );

    for(size_t i = 0; i < N; i ++){

        // precompute only c2 values
        for(size_t j = 0; j < input.N+2; j ++){
            buffer_c2[j] =
            row_input_current[j-input.distance_depth-input.distance_row] +
            row_input_current[j-input.distance_depth+input.distance_row];
        }

        // restriction
        for(size_t j = 1, J = 2; j < N+1; j++, J+=2){
            row_this_current[j] =
        }
c1 * ( row_input_current[J - input.distance_depth] ) +
c2 * ( row_input_current[J - input.distance_depth -1] +
    row_input_current[J - input.distance_depth +1] +
    buffer_c2[J]) +
c3 * ( buffer_c2[J-1] + buffer_c2[J+1]);
} // end for (restriction)

// next row
row_this_current += distance_row;
row_input_current += input.distance_row*2;
}

} // end if

void ParallelGrid3D::residual(ParallelGrid3D& input, SparsePoint* v, size_t
v_sz){
    // coefficients
    // c0 multiplies the grid value at (i,j,k)
    const double c0 = coeff_residual[0];
    #ifdef MG_RESID_ENABLE_C1
    // c1 multiplies the six values at grid points which differ by one in
    // exactly one index
    const double c1 = coeff_residual[1]; // c1 is zero in the NPB specification
    #endif
    // c2 multiplies the next closest twelve values, those that differ by one
    // in exactly two indices
    const double c2 = coeff_residual[2];
    // c3 multiplies the eight values located at grid point that differ by one
    // in all three indices
    const double c3 = coeff_residual[3];

    // buffer for temps
    double* restrict buffer_c2 = CACHE_ALIGNED(buffers);
    double* restrict buffer_c3 = CACHE_ALIGNED(buffers + distance_row);

    // pointer to current depth in the correction grid
    double* depth_correction_current = input.values + distance_depth;
    // pointer to current depth in this grid
    double* depth_this_current = values + distance_depth;

    // move by depth
    for(size_t k = 0; k < depth_length; k++){
        // skip first row as it contains boundary values
        double* restrict row_correction_current =
            CACHE_ALIGNED( depth_correction_current + distance_row );
        double* restrict row_this_current =
            CACHE_ALIGNED( depth_this_current + distance_row );

        // move by rows
        for(size_t i = 0; i < N; i++){
            // pos_correction_current = row_correction_current;
            for(size_t j = 0; j < N+2; j ++){
                buffer_c2[j] = row_correction_current[j -distance_depth] +
                                row_correction_current[j -distance_row] +
                                row_correction_current[j +distance_row] +
                                row_correction_current[j +distance_depth];
            }
        }
    }
}
for(size_t j = 0; j < N+2; j ++){
    buffer_c3[j] =
        row_correction_current[j -distance_depth -distance_row] +
        row_correction_current[j -distance_depth +distance_row] +
        row_correction_current[j +distance_depth -distance_row] +
        row_correction_current[j +distance_depth +distance_row];
}

for(size_t j = 1; j < N+1; j++){
    row_this_current[j] = - (c0 * row_correction_current[j] +
#ifdef MG_RESID_ENABLE_C1
    c1 * ( row_correction_current[j-1] +
        row_correction_current[j+1] +
        buffer_c2[j]) +
#endif
    c2 * ( buffer_c2[j -1] + buffer_c2[j +1] + buffer_c3[j]) +
    c3 * ( buffer_c3[j -1] + buffer_c3[j +1])
);
} // end for (residual)

// next row
    row_correction_current += distance_row;
    row_this_current += distance_row;
}

// move ahead to the next depth
    depth_correction_current += distance_depth;
    depth_this_current += distance_depth;
}

add(v,v_sz); // r = - A u --> r' = v - A u <= r' += v
update_local_boundaries();
}

void ParallelGrid3D::smoother(ParallelGrid3D& input){
    // Similar to add_smother(), replace the additions with the assignments
    // (...)
}

void ParallelGrid3D::subtract_residual(ParallelGrid3D& input){
    // Similar to residual(), edit r =- A u.
    // (...)
}

double ParallelGrid3D::get(int i, int j, int k) const{
    if(k < ((int)depth_start)-1 || k > (int) ((depth_start+depth_length) +1))
        throw std::invalid_argument("[Grid::get] \invalid\depth");
    return get0(i,j, k-depth_start);
}

void ParallelGrid3D::set(int i, int j, int k, double v){
    if(k < ((int)depth_start) || k > (int) (depth_start+depth_length))
        throw std::invalid_argument("[Grid::set] \invalid\depth");
    return set0(i,j,k-depth_start, v);
}

void ParallelGrid3D::add(SparsePoint* v, const size_t v_sz){
    for(size_t h = 0; h < v_sz; h++){
size_t k = v[h].k -1;

if(k > depth_start && k < depth_start + depth_length +1){
    incr0(v[h].i -1, v[h].j -1, k - depth_start, v[h].value);
}
}

void ParallelGrid3D::set(SparsePoint* v, const size_t v_sz){
for(size_t h = 0; h < v_sz; h++){
    size_t k = v[h].k -1;

    if(k > depth_start && k < depth_start + depth_length +1){
        set0(v[h].i -1, v[h].j -1, k - depth_start, v[h].value);
    }
}

void ParallelGrid3D::set_neighbours(ParallelGrid3D* previous_, ParallelGrid3D* next_){
if(!previous_ && !next_){
    neighbour_previous = neighbour_next = NULL;
} else if(!(previous_ || next_){
    /* throws an exception, both args must be ==NULL or !=NULL */
} else {
    // default
    if(previous_->N != N || next_->N != N){
        /* throws an exception, grids have different dimensions */
    } else {
        neighbour_previous = previous_;
        neighbour_next = next_;
    }
}

void ParallelGrid3D::update_local_boundary_body(){
    // move by depths
    double* mydepth = values + distance_depth;
    for(size_t k = 0; k < depth_length; k++){
        // move by rows
        double* myrow = mydepth + distance_row;
        for(size_t i = 0; i < N; i++){
            myrow[0] = myrow[N];
            myrow[N+1] = myrow[1];
            myrow += distance_row;
        }
        // copy the rows
        memcpy(mydepth, mydepth + distance_depth - (2*distance_row),
              sizeof(double) * (N+2));
        memcpy(mydepth + distance_depth - distance_row, mydepth + distance_row,
              sizeof(double) * (N+2));
        mydepth += distance_depth;
    }
}

void ParallelGrid3D::exchange_boundaries(){
if(!neighbour_previous){
    // local update
    memcpy(values, values + depth_length * distance_depth,
           sizeof(double) * (N+2));
}
    sizeof(double) * distance_depth);
    memcpy(values + (depth_length+1) * distance_depth,
            values + distance_depth, sizeof(double) * distance_depth);
} else { // exchange remotely
    memcpy(neighbour_previous->values + (neighbour_previous->depth_length+1)
            * distance_depth, values + distance_depth,
            sizeof(double) * distance_depth);
    memcpy(neighbour_next->values, values + depth_length * distance_depth,
            sizeof(double) * distance_depth);
}

void ParallelGrid3D::print_local_values(size_t internal_from, size_t internal_length) const{
    // with 0 : boundary above
    // 1 : depth_start
    // N+1 : depth_start + depth_length
    // N+2 : boundary below

    for(size_t k = internal_from; k < internal_from+internal_length; k++){
        cout << std::fixed << setprecision(4);
        // header
        cout << "[␣depth␣below:␣" << (depth_start+k) << '␣';
        if(k==0 || k == depth_length+1) { // is boundary?
            cout << "(boundary)␣");
            for(int t = 0; t < 60; t++) { cout << '-'; } cout << endl;
        } else {
            cout << "]";
            for(int t = 0; t < 72; t++) { cout << '-'; } cout << endl;
        }

        for(size_t i = 0; i < N+2; i++){ // rows
            for(size_t j = 0; j < N+2; j++){ // columns
                cout << get0(i,j,k) << '␣';
            }
            cout << '\n';
        } // end for (rows)
    } // end for (depth)
}

void ParallelGrid3D::erase(){
    memset(values, 0, distance_depth*(depth_length+2)*sizeof(double));
}

/**************************************************************************/
* STATIC METHODS
/**************************************************************************/
for(size_t i = 0; i < out_sz; i++){
    out[i]->exchange_boundaries();
}

void ParallelGrid3D::add_smoother(ParallelGrid3D* u_next[], ParallelGrid3D* u_prev[], const size_t sz){
    for(size_t i = 0; i < sz; i++){ u_next[i]->add_smoother(*u_prev[i]); } }

void ParallelGrid3D::interpolation(ParallelGrid3D* out[], const size_t out_sz, ParallelGrid3D* in[], const size_t in_sz){
    for(size_t i = 0; i < out_sz; i++){
        for(size_t j = 0; j < in_sz; j++){
            out[i]->interpolation(*in[j]);
        }
    }
    for(size_t i = 0; i < out_sz; i++){
        out[i]->exchange_boundaries();
    }
}

void ParallelGrid3D::interpolation_bounds(ParallelGrid3D* out_, ParallelGrid3D* in_, std::size_t* out_start_, std::size_t* in_start_, std::size_t* in_length_, bool* skip_first_, bool* skip_last_){
    long in0 = in_->depth_start *2; // input from
    long in1 = in0 + in_->depth_length *2; // input to

    // trick: for the output grid, the last depth would not be considered,
    // as it would expect that it is responsibility of the next partition,
    // i.e. the first, to provide the values. For this case it will exploit
    // the boundary of the last input grid.
    if(in1 == long(out_->N)) in1 += 2;

    long out0 = out_->depth_start +1; // output from
    long out1 = out0 + out_->depth_length; // output to
    long x0 = std::max(in0, out0); // left point in the shared interval
    long x1 = std::min(in1, out1); // right point in the shared interval
    long out_length = x1 - x0; // length in terms of out
    bool skip_first = x0 % 2 == 1;
    bool skip_last = x1 % 2 == 1;

    *in_start_ = static_cast<size_t>(std::max((out0 - in0)/2, 0l));
    *out_start_ = static_cast<size_t>(std::max(in0 - out0, 0l)) +1;
    *in_length_ = static_cast<size_t>(
        std::max(long(ceil(double(out_length)/2)), 0l));
    if(out_length > 0 && skip_first && skip_last) (*in_length_++)
    *skip_first_ = skip_first;
    *skip_last_ = skip_last;
}

double ParallelGrid3D::normL2(ParallelGrid3D* grids[], size_t grids_sz){
    double sum(0);
    double total_points(0); // in theory it should be N^3

    // move by grids
    for(size_t g = 0; g < grids_sz; g++){
double* restrict current_depth =
grids[g]->values + grids[g]->distance_depth;
total_points += pow(grids[g]->N, 2.) * grids[g]->depth_length;

// move by depth
for(size_t k = 0; k < grids[g]->depth_length; k++){
    // move by rows
    double* current_row = current_depth + grids[g]->distance_row;
    for(size_t i = 0; i < grids[g]->N; i++){
        // move by columns
        for(size_t j = 1; j < (grids[g]->N)+1; j++){
            sum+= pow(current_row[j] ,2.);
        }
        current_row += grids[g]->distance_row;
    }
    current_depth += grids[g]->distance_depth;
}

if(!total_points){ // avoid division by zero
    return 0.;
} else {
    return sqrt(sum/total_points);
}
}

void ParallelGrid3D::print_values(ParallelGrid3D* grids[], size_t grids_sz){
    if(grids_sz == 0) return;
    if(grids_sz == 1) return grids[0]->print_local_values();

    size_t N = grids[0]->N;
    for(size_t i = 0; i < grids_sz; i++){
        if(grids[i]->N != N ) { // throws an exception (... ) */
            if(i==0){ // first grid
                grids[i]->print_local_values(0, grids[i]->depth_length+1);
            } else if(i == grids_sz -1){ // last grid
                grids[i]->print_local_values(1, grids[i]->depth_length+1);
            } else { // middle grids
                grids[i]->print_local_values(1, grids[i]->depth_length);
            }
        }
    }
}

void ParallelGrid3D::residual(ParallelGrid3D* r[], ParallelGrid3D* u[], std::size_t sz, SparsePoint* v, std::size_t v_sz){
    for(size_t i = 0; i < sz; i++){ r[i]->residual(*u[i], v, v_sz); }
}

void ParallelGrid3D::restriction_bounds(ParallelGrid3D* out_, ParallelGrid3D* in_, std::size_t* out_start_, std::size_t* out_length_, std::size_t* in_start_, bool* skip_first_, bool* skip_last_){
    long in0 = in_->depth_start +1; // input from
long in1 = in0 + in_->depth_length; // input to

// same trick as interpolation_bounds
if(in1 == long(in_->N) +1) in1++;

long out0 = (out_->depth_start +1)*2; // output from
long out1 = out0 + out_->depth_length*2; // output to
long x0 = std::max(in0, out0); // left point in the shared interval
long x1 = std::min(in1, out1); // right point in the shared interval
long in_length = x1 - x0; // length in terms of out
bool skip_first = x0 % 2 == 1;
bool skip_last = x1 % 2 == 1;

*out_start_ = static_cast<size_t>(std::max((in0 - out0)/2, 0l)) +1;
in_start_ = static_cast<size_t>(std::max(out0 - in0, 0l));
if(!skip_first) *in_start_ += 1; // do not exploit the first boundary depth
*out_length_ = static_cast<size_t>(
    std::max(long(ceil(double(in_length)/2)), 0l));
if(in_length > 0 && skip_first && skip_last) (*out_length_++) = 1;
*skip_first_ = skip_first;
*skip_last_ = skip_last;
}

void ParallelGrid3D::restriction(ParallelGrid3D* out[], const size_t out_sz, ParallelGrid3D* in[], const size_t in_sz)
{
    for(size_t i = 0; i < out_sz; i++)
    {
        for(size_t j = 0; j < in_sz; j++)
        {
            out[i]->restriction(*in[j]);
        }
    }
}

void ParallelGrid3D::set(ParallelGrid3D* grids[], size_t grids_sz, int i, int j, int k, double v)
{
    bool is_set = false;
    size_t index = 0;
    while(index < grids_sz && !is_set)
    {
        if(k > (int)(grids[index]->depth_start) && k < (int)(grids[index]->depth_start + grids[index]->depth_length +1))
        {
            grids[index]->set0(i, j, k - grids[index]->depth_start, v);
            is_set = true;
        } else {
            index++;
        }
    }

    if(!is_set) { /* throws an exception (...) */ }
}

void ParallelGrid3D::set(ParallelGrid3D* grids[], size_t grids_sz, SparsePoint v[], size_t v_sz)
{
    // as the v points are at most 20, this approach should still be effective.
    for(size_t i = 0; i < grids_sz; i++) grids[i]->set(v, v_sz);
}
Parallel algorithm

The listings below show the implementation of the parallel algorithm. The basic scheme is similar to the CG algorithm: a Master and a set of Workers (Slaves) cooperate to advance in the computation. The atomic counter current_active_workers ensures the synchronisation between the two types of entities. The code provides a custom Scheduler, similar to the algorithm CG, to select the worker, the one with ID=0, that carries out the sequential part of the algorithm. The PartitionDatabase is a
container to record the partitions related to workers. Finally, a user can specify the strategy to pin
the physical threads:

- interleaved: the default strategy, the algorithm selects one physical thread per node in an
  interleaved manner. The presented simulations exploited this strategy.

- fill: the algorithm attempts to get all the physical threads from the same node, and moves to
  a different node when there are not more available threads. The code refers to this strategy as
  *NumaGrid*. The experiment in figure 5.9 used this strategy.

Listing A.14: MG / parallel.h

```cpp
1 // classes defined in this header
2 class Master;
3 class Parallel;
4 class PartitionDatabase;
5 class PartitionDescriptor;
6 class Scheduler;
7 class Slave;

8 class Parallel : public my::mg::Base {
9     protected:
10         friend class Master;
11         friend class Slave;
12
13         /**
14         * The number of levels for this system, i.e. log2(n).
15         */
16         const int mg_levels;
17
18         /**
19         * Use the numa impl. VS the automatic topology
20         */
21         const bool numagrid;
22
23         // Container for the workers
24         typedef std::vector<ff::ff_node*> workers_t;
25         workers_t workers;
26         const int num_workers;
27
28         // Farm
29         ff::ff_farm<Scheduler>* farm;
30         Master* master;
31         //Collector* collector;
32
33         // synchronise workers with the master
34         atomic_t current_active_workers;
35
36         // Records the created grids
37         PartitionDatabase* partitions;
38
39         // perform the actual algorithm
40         protected: virtual void execute();
41     }
42
43 public:
44
45         /**
46         * Default constructor
47         */
```

134
```cpp
Parallel();

/**
 * Destructor
 */
virtual ~Parallel();

/**
 * SCHEDULER
 */

/* Scheduler of the tasks among the workers.
 * If pick_sequential_next() has been invoked, then it will return as next
 * worker the node implementing the sequential part of the multigrid kernel.
 * Otherwise it returns the workers according to the round-robin policy.
 */
class Scheduler: public ff::ff_loadbalancer{
protected:
    int num_workers; // number of available workers = parallelism grade
    int node_id_sequential; // id of the super worker
    int next_worker; // the next worker id to return
    bool flag_pick_master; // flag to select the super worker

    /**
     * Contract -- select the next worker according to the scheduling policy
     */
    inline int selectworker(){
        if(flag_pick_master){
            next_worker = 0;
            flag_pick_master = false;
            return node_id_sequential;
        } else { // round robin
            int current_worker = next_worker;
            next_worker = (next_worker + 1) % num_workers;
            return current_worker;
        }
    }

public:
    /**
     * Constructor
     * @param num_workers: the number of available workers
     * @param node_sequential: id of the worker that will execute the
     * sequential part of the algorithm
     */
    Scheduler(int num_workers_, int node_sequential_ =0) : ff_loadbalancer(num_workers_),
        num_workers(1),
        node_id_sequential(node_sequential_),
        next_worker(0),
        flag_pick_master(false){ }

    /**
     * Sets the number of available workers
     */
    void set_size_workers(int num){ num_workers = num; }
```
```cpp
void set_sequential_worker(int node_id){ node_id_sequential = node_id; }

void pick_sequential_next(){ flag_pick_master = true; }; // class Scheduler

class PartitionDatabase{
private:
    typedef boost::mutex mutex_t;
    typedef boost::lock_guard<mutex_t> lock_t;
    typedef std::map<int, NumaGrid3D*> node_dictionary_t;

    int cutoff; // cutoff threshold for the sequential algorithm
    int num_workers; // total amount of workers
    int num_levels; // number of levels

    ParallelGrid3D*** corrections; // array of levels of created partitions
    ParallelGrid3D*** residuals;

    struct pthread_t{int thread_id; int node_id;};
    pthread_t* pthreads; // threads used by the workers

    node_dictionary_t nodegrids;
    mutex_t mutex;

    int map2grid(int worker_id, int level, int shift);

public:

    /* Constructor */
    PartitionDatabase(int cutoff, int num_workers, int num_levels, std::vector<int>* available_threads = NULL);

    /* Destructor */
};
```
/**  
* Sets the record for a certain partition  
*/  
void set(int id, int lvl, ParallelGrid3D* residual, ParallelGrid3D* correction){  
    residuals[lvl][id] = residual;  
    corrections[lvl][id] = correction;  
}  
  
/**  
* Accessors to the employed physical threads  
*/  
int get_position(int thread_id);  
int get_thread(int worker_id);  
int get_node(int worker_id);  
  
/**  
* Used only in the numa impl.  
*/  
// note that #workers >= #nodes  
NumaGrid3D* get_nodegrid_correction(int worker_id, int level){ lock_t lock(mutex); return nodegrids[map2grid(worker_id, level, 0)];}  
void set_nodegrid_correction(int worker_id, int level, NumaGrid3D* grid){  
    lock_t lock(mutex);  
    nodegrids[ map2grid(worker_id, level, 0) ] = grid; }  
  
NumaGrid3D* get_nodegrid_residual(int worker_id, int level){lock_t lock(mutex); return nodegrids[ map2grid(worker_id, level, 1) ]; };  
void set_nodegrid_residual(int worker_id, int level, NumaGrid3D* grid){  
    lock_t lock(mutex);  
    nodegrids[ map2grid(worker_id, level, 1) ] = grid; };  
  
/**  
* Retrieves the list of residuals/corrections for a given level.  
*/  
void get_residuals(int lvl, ParallelGrid3D*** ptr_grids, int* ptr_grids_sz){  
    *ptr_grids = residuals[lvl];  
    *ptr_grids_sz = lvl<= cutoff? 1 : num_workers;  
}  
void get_corrections(int lvl, ParallelGrid3D*** ptr_grids, int* ptr_grids_sz){  
    *ptr_grids = corrections[lvl];  
    *ptr_grids_sz = lvl<= cutoff? 1 : num_workers;  
}  
};  
  
/**  
* Wrapper. Based on its id, tells to a specific worker which partitions  
* to own.  
*/  
class PartitionDescriptor{  
private:  
    std::size_t N; // dimension of a partition  
    std::size_t start; // starting depth  
    std::size_t length; // length of the partition  
    std::size_t level; // its level, i.e. log2(N) –1.  
  
public:  
    /**<  
    * Accessors  
    */

std::size_t get_N(){ return N; }
std::size_t get_start(){ return start; }
std::size_t get_length(){ return length; }
std::size_t get_level(){ return level; }

public:

/∗∗
∗ Creates the set of partitions that a worker must own.
∗ @param worker_id: id of the worker
∗ @param num_workers: total number of workers
∗ @param cutoff: cutoff threshold
∗ @param num_levels: total number of levels
∗ @param out_partitions: output argument, it will contain the array of
∗    partition descriptions
∗ @param out_partitions_sz: output value, the size of the array
∗ @param out_partitions_sz:
∗ /
static void create(int worker_id, int num_workers, int cutoff, int
num_levels, PartitionDescriptor** out_partitions, int* out_partitions_sz);

/∗∗
∗ Deallocates the array of partitions previously made with the method
∗ create();
∗ /
static void free_array(PartitionDescriptor partitions[]);

/********************************************************************************
∗ TASKS
********************************************************************************

/∗∗
∗ Possible tasks that workers may execute
∗ /
enum Task_Operation{
    TASK_INIT_NODES, // only numa impl.
    TASK_INIT_PARTITIONS,
    TASK_END,
    TASK_GO_SEQUENTIAL,
    // single operation
    TASK_ADD_INTERPOLATION,
    TASK_ADD_Smoother,
    TASK_INTERPOLATION,
    TASK_RESIDUAL,
    TASK_RESTRICTION,
    TASK_SET_V,
    TASK_Smoother,
    TASK_SUBTRACT_RESIDUAL,
    // global
    TASK_NORM2
};

/**
 ∗ Generic task
 */
struct Task{
Task_Operation op;
  int level;
};

/****************************************************************************
   * MASTER
   *
****************************************************************************/

/** Emitter module for the underlying farm paradigm */
class Master: public ff::ff_node{
protected:
  enum MasterState{ INIT, SET_ERROR, RESTRICTION, INTERPOLATION, RESIDUAL,
                   SMOOTHING, EXT_RESIDUAL, NORM2, END };
  Parallel& parallel; // reference to the kernel object
  const int cpu_id; // cpu where to pin the emitter
  const int num_workers; // total number of workers
  const int num_levels; // number of levels
  const int num_iterations; // how many iterations to compute
  const int lvl_cutoff; // the threshold where the algorithm will
                        // continue sequentially
  const bool send_init_nodes; // send the task INIT_NODES before
                              // than INIT_PARTITION
  MasterState current_state; // next operation to perform
  int current_level; // next level to consider
  int current_iteration;
  atomic_t& waiting_for_tasks; // track how many workers are still
                               // working
  Task task; // task shared with the workers
  Scheduler& scheduler; // reference to the farm scheduler
protected:
  inline void _send(){ ff_send_out(static_cast<Task*>(&task)); }
  inline void _send_all(){ for(int i = 0; i < num_workers; i++) { _send(); }}
  inline void _send_one(){ scheduler.pick_sequential_next(); _send(); }
  inline void _wait(){ while(atomic_read(&waiting_for_tasks)){} }
  inline void execute_all(){
    atomic_set(&waiting_for_tasks, num_workers); _send_all(); _wait();
  }
  inline void execute_sequential(){
    atomic_set(&waiting_for_tasks, 1); _send_one(); _wait();
  }
public:
/**
 * Constructor
 */
 Master(int cpu_id, int cutoff, Scheduler& scheduler, Parallel& parallel);

/**
 * Destructor
 */
 virtual ~Master();
/**
 * Pins the master to the specified cpu_id
 */
 virtual int svc_init();

/**
 * Contract — implements the logic of the emitter
 */
 virtual void* svc(void*);

//******************************************************************************
 * SLAVE
//******************************************************************************

/**
 * Workers for the multigrid farm
 */
class Slave : public ff::ff_node{
 protected:
 Parallel& parallel; // wrapper instance
 const int worker_id; // id of this worker
 const int cpu_id; // id of the related physical thread
 const int num_levels; // #levels in the computation
 const int sequential_cutoff; // cutoff threshold
 atomic_t& waiting_for_tasks; // sync counter
 ParallelGrid3D** residuals; // owned partitions
 ParallelGrid3D** corrections;
 double* buffer; // owned buffer, for ParallelGrid3D
 bool init_node; // flag, has the node been initialised?

 /**
 * Create the node memory chunks
 */
 protected: virtual void initialise_node();

 /**
 * Create the partitions according to the partition descriptor
 */
 protected: virtual void initialise_partitions();

 /**
 * Remove created partitions
 */
 protected: virtual void destroy();

 public:
 /**
 * Constructor
 * @param worker_id id of this worker
 * @param cpu_id id of the related physical thread
 * @param cutoff sequential cutoff threshold
 * @param parallel reference to the Farm wrapper
 */
Slave(int worker_id, int cpu_id, int cutoff, Parallel& parallel);

/**
 * Destructor
 */
~Slave();

/**
 * Pins this worker to the specified cpu_id
 */
virtual int svc_init();

/**
 * Performs the computation
 */
virtual void* svc(void*);

Listing A.15: MG / parallel.cpp

using namespace ff;
using std::size_t;

Parallel::Parallel() : Base(),
    mg_levels(log2(mg_size)),
    numagrid(get_configuration().get("parallel.numagrid", false)),
    num_workers(get_configuration().get_parallelism_grade()) {
    // parse args
    int cutoff = get_configuration().get<int>("parallel.cutoff");
    if(cutoff < 1){ /* throws an exception (...) */ }
    if(cutoff >= mg_levels -1){  /* throws an exception (...) */ }

    // atomic counter employed as synchroniser
    atomic_set(&current_active_workers, 0);

    // create the farm
    farm = new ff_farm<Scheduler>();
    Scheduler& scheduler = *((Scheduler*) farm->getlb());
    scheduler.set_size_workers(num_workers);
    scheduler.set_sequential_worker(0);

    // set the master
    master = new Master(0, cutoff, scheduler, *this);
    farm->add_emitter(master);

    // create the partitions
    std::vector<int> threadmapping;
    cpu_topology::threadlist(threadmapping, 0,
        numagrid ? cpu_topology::MODE_FILL : cpu_topology::MODE_MAX_AVAIL);
    partitions = new PartitionDatabase(cutoff, num_workers, mg_levels,
        (numagrid ? &threadmapping : NULL));

    // create the workers
    for(int i = 0; i < num_workers; i++){
        workers.push_back(new Slave(i, threadmapping[i], cutoff, *this));
    }

    // record the workers

farm->add_workers(workers);

Parallel::~Parallel() {
  delete master; master = NULL;
  // the farm deallocates the workers
  delete farm; farm = NULL;
  delete partitions; partitions = NULL;
}

void Parallel::execute() {
  // create the task
  if(farm->run_and_wait_end() < 0) {
    /* throws an exception (...) */
  }
}

PartitionDatabase::PartitionDatabase(int cutoff_, int num_workers_, int num_levels_, std::vector<int>* available_threads_) :
  cutoff(cutoff_), num_workers(num_workers_), num_levels(num_levels_) {
    // make space for partitions
    corrections = new ParallelGrid3D**[num_levels_];
    residuals = new ParallelGrid3D**[num_levels_];

    for(int i = 0; i <= cutoff_; i++) {
      corrections[i] = new ParallelGrid3D*[1];
      residuals[i] = new ParallelGrid3D*[1];
    }

    for(int i = cutoff_ + 1; i < num_levels_; i++) {
      corrections[i] = new ParallelGrid3D*[num_workers];
      residuals[i] = new ParallelGrid3D*[num_workers];
    }

    if(available_threads_) {
      // threads are mapped continuously in a numa node
      if(available_threads_->size() < size_t(num_workers)) {
        /* throws an exception (...) */
      }
    }

    // record the available nodes/threads
    pthreads = new pthread_t[num_workers_];
    cpu_topology& topo = get_cpu_topology();
    for(int i = 0; i < num_workers; i++) {
      int thread_id = available_threads_->at(i);
      pthreads[i].thread_id = thread_id;
      pthreads[i].node_id = topo.get_node(thread_id);
    }
  } else {
    // do not exploit this feature
    pthreads = NULL;
  }
PartitionDatabase::~PartitionDatabase(){
    for(int i = 0; i < num_levels; i++){
        delete[] corrections[i]; corrections[i] = NULL;
        delete[] residuals[i]; residuals[i] = NULL;
    }
    delete[] corrections; corrections = NULL;
    delete[] residuals; residuals = NULL;
}

delete[] pthreads; pthreads = NULL;
}

int PartitionDatabase::get_position(int thread_id){
    if(!pthreads) { /* throws an exception (...) */ }
    for(int i = 0; i < num_workers; i++){
        if(pthreads[i].thread_id == thread_id) return i;
    }
    // not found, throws an exception (...)
}

int PartitionDatabase::get_thread(int worker_id){
    if(!pthreads) { /* throws an exception (...) */ }
    return pthreads[worker_id].thread_id;
}

int PartitionDatabase::get_node(int worker_id){
    if(!pthreads) { /* throws an exception (...) */ }
    return pthreads[worker_id].node_id;
}

int PartitionDatabase::map2grid(int worker_id, int level, int shift){
    if(!pthreads) { /* throws an exception (...) */ }
    return 2*(level*num_workers + pthreads[worker_id].node_id) +shift;
}

/* ************************************************* *
 * Partition Descriptor
 * ************************************************* */

void PartitionDescriptor::create(const int worker_id_, const int num_workers_,
    const int cutoff_, const int num_levels_,
    PartitionDescriptor** a_, int* a_sz_)
{
    int a_sz = (worker_id_ == 0) ? num_levels_ : num_levels_ - (cutoff_+1);
    PartitionDescriptor* a = new PartitionDescriptor[a_sz];
    int index = 0; // current index in the array
    size_t worker_id = static_cast<size_t>(worker_id_);

    // sequential
    if(!worker_id){
        for(int lvl = 0; lvl <= cutoff_; lvl++){
            a[index].N = pow(2,lvl+1);
        }
    }
}
```cpp
158  a[index].start = 1;
159  a[index].length = a[index].N; // default
160  a[index].level = size_t(lvl);
161
162  index++;
163 }
164 }
165
166 // parallel
167 for(int lvl = cutoff_ +1; lvl < num_levels_; lvl++){
168   size_t N = pow(2, lvl+1);
169   size_t div = N / num_workers_;  
170   size_t rem = N % num_workers_;   // remainder
171   size_t start = (div * worker_id) + std::min(worker_id, rem) +1;
172   size_t length = (worker_id < rem) ? div +1 : div;
173
174   a[index].N = N;
175   a[index].start = start;
176   a[index].length = length;
177   a[index].level = size_t(lvl);
178   index++;
179 }
180
181
182  *a_ = a;
183  *a_sz_ = a_sz;
184 }
185
v3d PartitionDescriptor::free_array(PartitionDescriptor a[]){ delete[] a; }
186
187 //**********************************************************************
188 /*  
189  */
190 /*  
191  MASTER 
192  */
193 //**********************************************************************
194 Master::Master(int cpu_id_, int cutoff_, Scheduler& scheduler_, Parallel& parallel_ ) :
195   parallel(parallel_), cpu_id(cpu_id_), num_workers(parallel.num_workers),
196   num_levels(log2(parallel.mg_size)), num_iterations(parallel.mg_iterations),
197   lvl_cutoff(cutoff_), send_init_nodes(parallel.numagrid),
198   current_state(INIT), current_level(0),current_iteration(0),
199   waiting_for_tasks(parallel.current_active_workers), task(),
200   scheduler(scheduler_)
201 { }
202
203 Master::~Master(){ }
204
205 int Master::svc_init(){ ff_mapThreadToCpu(cpu_id); return 0; }
206
207 void* Master::svc(void* args){
208   using std::cout;
209   using std::flush;
210   using std::endl;
211
212   while(current_state!=END){
213     switch(current_state){
214       case INIT:
215 ```
{  
  if(send_init_nodes){ // init NumaGrid3D for each node
    task.op = TASK_INIT_NODES;
    execute_all();
  }

  task.op = TASK_INIT_PARTITIONS;
  execute_all();

  // link the partitions
  for(int i = 0; i < num_levels; i++){
    ParallelGrid3D** a; int a_sz; // temp
    parallel.partitions->get_residuals(i, &a, &a_sz);
    ParallelGrid3D::set_circular_neighbours(a, a_sz);
    parallel.partitions->get_corrections(i, &a, &a_sz);
    ParallelGrid3D::set_circular_neighbours(a, a_sz);
  }

  current_level = num_levels -1;
  current_state = SET_ERROR;
  current_iteration = 1;

  // let's start!!
  parallel.timer.start();

  break;
}
case SET_ERROR:
{
  task.op = TASK_SET_V;
  task.level = current_level;

  execute_all();

  current_iteration = 1;
  current_state = RESTRICTION;
  current_level--;

  break;
}
case RESTRICTION:
{
  task.level = current_level;
  
  if(current_level > lvl_cutoff){
    task.op = TASK_RESTRICTION;
    execute_all();

    // current_state = RESTRICTION
    current_level--;
  } else { // continue sequentially
    task.op = TASK_GO_SEQUENTIAL;
    execute_sequential();

    current_state = INTERPOLATION;
    current_level++;
  }
break;

}  

case INTERPOLATION:
{
    // common params
    task.op = (current_level < num_levels -1) ?
        TASK_INTERPOLATION : TASK_ADD_INTERPOLATION;
    task.level = current_level;  // should be untouched
    execute_all();
    current_state = RESIDUAL;
    // same level
    break;
}

case RESIDUAL:
{
    task.op = (current_level < num_levels -1) ?
        TASK_SUBTRACT_RESIDUAL : TASK_RESIDUAL;
    task.level = current_level;  // redundant
    execute_all();
    current_state = SMOOTHING;
    break;
}

case SMOOTHING:
{
    task.op = (current_level > 0) ? TASK_ADD_SMOOTHER : TASK_SMOOTHER;
    task.level = current_level;  // redundant
    execute_all();
    current_level++;
    if(current_level >= num_levels){
        current_state = EXT_RESIDUAL;
        current_level = num_levels -1;
    } else {
        current_state = INTERPOLATION;
    }
    break;
}

case EXT_RESIDUAL:
{
    task.op = TASK_RESIDUAL;
    task.level = current_level;  // redundant?
    execute_all();
    current_iteration++;
    if(current_iteration <= num_iterations){
        current_state = RESTRICTION;
        current_level--;  
    } else { // ended
        current_state = NORM2;
    }
}
break;
}

case NORM2:
{
    task.op = TASK_NORM2;
    task.level = num_levels -1;
    execute_sequential();
    parallel.timer.stop();
current_state = END;

break;
}
default:
    throw std::runtime_error("[Master::svc] unrecognised status");
}  // end switch
}  // end if

// finalise
task.op = TASK_END;
execute_all();
current_state = INIT;  // if the user requests to restart the computation.
return NULL;  // stop the framework

void Slave::initialise_node(){
    // is my responsibility to initialise the node?
if(! (parallel.numagrid && ( worker_id == 0 ||
    ( parallel.partitions->get_node(worker_id -1) !=
    parallel.partitions->get_node(worker_id)) )))
{
    return;
}
init_node = true;
const double* cf_residual = parallel.coeff_residual;
const double* cf_smoothers = parallel.coeff_smoothing;

// am I the sequential worker?
if(!worker_id)
    for(int level = 0; level <= sequential_cutoff; level++) {
        size_t N = pow(2, level+1);
        parallel.partitions->set_nodegrid_correction(worker_id, level,
           new NumaGrid3D(N, cf_residual, cf_smoothers));
        parallel.partitions->set_nodegrid_residual(worker_id, level,
           new NumaGrid3D(N, cf_residual, cf_smoothers));
    }

const size_t num_workers = static_cast<size_t>(parallel.num_workers);

// nodes
for(int level = sequential_cutoff+1; level < num_levels; level++) {
    // partition info
    size_t N = pow(2, level+1);
    size_t div = N / num_workers;
    size_t rem = N % num_workers; // remainder
    size_t id = worker_id;
    size_t start = (div * id) + std::min(id, rem) +1;
    size_t length = 0;
    size_t node_id = parallel.partitions->get_node(worker_id);
    do{
        length += (id < rem) ? div +1 : div;
        id++;
    } while(id < num_workers &&
            size_t(parallel.partitions->get_node(id)) == node_id);

    parallel.partitions->set_nodegrid_correction(worker_id, level,
       new NumaGrid3D(N, cf_residual, cf_smoothers, start, length));
    parallel.partitions->set_nodegrid_residual(worker_id, level,
       new NumaGrid3D(N, cf_residual, cf_smoothers, start, length));
}

void Slave::initialise_partitions() {
    // init the buffer
    buffer = ParallelGrid3D::buffer_alloc(parallel.mg_size);

    // some of these pointers will remain NULL
    residuals = new ParallelGrid3D*[num_levels]();
    corrections = new ParallelGrid3D*[num_levels]();
    PartitionDescriptor* partitions; int partitions_sz;
    PartitionDescriptor::create(worker_id, parallel.num_workers,
       sequential_cutoff, num_levels, &partitions, &partitions_sz);

    // init the partitions
    for(int i = 0; i < partitions_sz; i++) {
        const int l = partitions[i].get_level(); // level
        const size_t N = partitions[i].get_N(); // implicit
        const size_t start = partitions[i].get_start();
        const size_t length = partitions[i].get_length();

        if(parallel.numagrid) // numa impl, November 2nd, 2012
            residuals[l] = parallel.partitions->
               get_nodegrid_residual(worker_id, l)->
grid(start, length, buffer);
corrections[l] = parallel.partitions->
    get_nodegrid_correction(worker_id, l)->
    grid(start, length, buffer);
} else { // previous impl. October 31, 2012
    residuals[l] = new ParallelGrid3D(N, parallel.coeff_residual,
        parallel.coeff_smoothing, buffer, start, length);
corrections[l] = new ParallelGrid3D(N, parallel.coeff_residual,
        parallel.coeff_smoothing, buffer, start, length);
}

// register the partition in the kernel object
parallel.partitions->set(worker_id, l, residuals[l], corrections[l]);
}
PartitionDescriptor::free_array(partitions);
}

void Slave::destroy(){
    if(residuals){
        for(int i = 0; i < num_levels; i++){
            delete residuals[i]; residuals[i] = NULL;
        }
        delete[] residuals; residuals = NULL;
    }

    if(corrections){
        for(int i = 0; i < num_levels; i++){
            delete corrections[i]; corrections[i] = NULL;
        }
        delete[] corrections; corrections = NULL;
    }

    if(buffer){ ParallelGrid3D::buffer_free(buffer); buffer = NULL; }

    if(init_node){ // deallocate node partitions
        for(int l = (!worker_id) ? 0 : sequential_cutoff+1; l < num_levels; l++){
            NumaGrid3D* correction =
                parallel.partitions->get_nodegrid_correction(worker_id, l);
            delete correction; correction = NULL;
            parallel.partitions->set_nodegrid_correction(worker_id, l, NULL);

            NumaGrid3D* residual =
                parallel.partitions->get_nodegrid_residual(worker_id, l);
            delete residual; residual = NULL;
            parallel.partitions->set_nodegrid_residual(worker_id, l, NULL);
        } // end for

        // remind partitions have been removed
        init_node = false;
    } // end if
}

int Slave::svc_init(){ ff_mapThreadToCpu(cpu_id); return 0; }

void* Slave::svc(void* input){
    Task* task = static_cast<Task*>(input);
    const int l = task->level;

switch(task->op) {
    case TASK_INIT_NODES:
        initialise_node();
        break;
    case TASK_INIT_PARTITIONS:
        initialise_partitions();
        break;
    case TASK_GO_SEQUENTIAL:
        // args
        ParallelGrid3D** args; int args_sz;
        parallel.partitions->get_residuals(l+1, &args, &args_sz);

        // down cycle
        residuals[l]->_restriction(args, args_sz);
        for(int i = l-1; i >= 0; i--){
            residuals[i]->_restriction(*residuals[i+1]);
            residuals[i]->_update_local_boundaries();
        }

        corrections[0]->_smoother(*residuals[0]);

        // upper cycle
        for(int i = 1; i <= l; i++){
            corrections[i]->_interpolation(*corrections[i-1]);
            corrections[i]->_exchange_boundaries();
            residuals[i]->_subtract_residual(*corrections[i]);
            corrections[i]->_add_smoother(*residuals[i]);
        }

        break;
    case TASK_ADD_INTERPOLATION:
        ParallelGrid3D** args; int args_sz;
        parallel.partitions->get_corrections(l-1, &args, &args_sz);

        corrections[l]->_add_interpolation(args, args_sz);
        break;
    case TASK_ADD_SMOOTHER:
        corrections[l]->_add_smoother(*residuals[l]);
        break;
    case TASK_INTERPOLATION:
        ParallelGrid3D** args; int args_sz;
        parallel.partitions->get_corrections(l-1, &args, &args_sz);

        corrections[l]->_interpolation(args, args_sz);
        break;
    case TASK_RESIDUAL:
        residuals[l]->_residual(*corrections[l], parallel.constant_terms,
    parallel.constant_terms_sz);
    break;
}

    case TASK_RESTRICTION:
    {
        ParallelGrid3D** args; int args_sz;
        parallel.partitions->get_residuals(l+1, &args, &args_sz);
        residuals[l]->restriction(args, args_sz);
        break;
    }

    case TASK_SET_V:
    {
        residuals[l]->set(parallel.constant_terms, parallel.constant_terms_sz);
        residuals[l]->update_local_boundaries();
        break;
    }

    case TASK_SMOOTHER:
    {
        corrections[l]->smoother(*residuals[l]);
        break;
    }

    case TASK_SUBTRACT_RESIDUAL:
    {
        residuals[l]->subtract_residual(*corrections[l]);
        break;
    }

    case TASK_NORM2:
    {
        ParallelGrid3D** args; int args_sz;
        parallel.partitions->get_residuals(l, &args, &args_sz);
        double norm2 = ParallelGrid3D::normL2(args, args_sz);
        parallel.set_computed_norm_l2(norm2);
        break;
    }

    case TASK_END:
    {
        destroy();
        break;
    }

    default:
    {
        throw std::runtime_error("[Slave::svc] Unsupported task operation");
    }
}

    atomic_dec(&waiting_for_tasks);
    return GO_ON;
}
References

tions of the NAS kernel benchmarks on the IBM SP2. IBM Systems

cient streaming on multi-core. Programming multi-core and many-core com-

TR-12-04, University of Pisa. 2012.


[9] Burkardt J. RANDLC: NAS parallel benchmark pseudo-random num-
sc.fsu.edu/~jburkardt/c_src/randlc/randlc.html

kernels. Project’s homepage: http://threads.hpcl.gwu.edu/sites/npb-upc

Models. Department of Informatics, University of Pisa. 2012.


[22] Jin H, Frumkin M, Yan J. The OpenMP implementation of NAS Parallel

[23] Kai C++ compiler. User’s guide: helping the optimizer. Available at:
http://www-d0.fnal.gov/KAI/doc/UserGuide/chapter_4.html

Pearson, Prentice-Hall. 2004


[26] NAS Parallel Benchmarks, project’s homepage. Available at: http://
www.nas.nasa.gov/publications/npb.html

www2.opengroup.org/ogsyst/catalog/c082

[28] Saini S, Bailey D. NAS parallel benchmark (version 1.0) results 11-96. NAS

results for the NAS Parallel Benchmarks 2. In Proceedings of the 8th SIAM

Press. 2007.


[32] Vanneschi M. Lecture notes of the course High Performance Comput-
ing. Department of Informatics, University of Pisa. 2012. Available at:
http://www.di.unipi.it/~vannesch/