Parallel Algebraic Linear Algebra Dedicated Interface[*]

Thierry Gautier
LIG-MOAIS UJF, CNRS, Inria, G'INP, UPMF
Inovallée, 655, av. de l'Europe, F38334 St Ismier Cedex, France
thierry.gautier@inrialpes.fr

Jean-Louis Roch
LIG-MOAIS UJF, CNRS, Inria, G'INP, UPMF
Inovallée, 655, av. de l'Europe, F38334 St Ismier Cedex, France
Jean-Louis.Roch@imag.fr

Ziad Sultan
Univ. Grenoble Alpes
Laboratories LJK and LIG
Inria, CNRS, Univ. Grenoble Alpes
Inovallée, 655, av. de l'Europe, F38334 St Ismier Cedex, France
Ziad.Sultan@imag.fr

Bastien Vialla
Université Montpellier
LIRMM, CNRS, 161 rue Ada, F-34095 Montpellier, France
Bastien.Vialla@lirmm.fr

ABSTRACT

This work deals with parallelism in linear algebra routines. We propose a domain specific language based on C/C++ macros, PALADIn (Parallel Algebraic Linear Algebra Dedicated Interface). This domain specific language allows the user to write C++ code and benefit from sequential and parallel executions on shared memory architectures. With a unique syntax, the user can switch between different parallel runtime systems such as OpenMP, TBB and xKaapi. This interface provides data and task parallelism. Depending on the runtime system, task parallelism can uses explicit synchronizations or data-dependency based synchronizations. Also, this language provides different matrix cutting strategies according to one or two dimensions. Moreover, block algorithms, such as block iterative and recursive matrix multiplication, can involve splitting according to three dimensions. The latter is also a feature that is provided to the user. The PALADIn interface can be used in any C++ library for linear algebra computation and gets the best performance from the three supported parallel runtime systems.

Keywords

Domain Specific Language, Shared Memory Parallelism, Dataflow Parallelism, Cutting Strategy

1. INTRODUCTION

Recently many efforts have been made to obtain efficient parallel implementation of linear algebra libraries. Sequential generic libraries exist to solve exact [7, 14, 6] and numerical [1, 15] linear algebra problems. As for parallel computing many libraries exist in numerical linear algebra [19, 15], whereas in exact computation there is not yet a parallel generic library.

To solve exact linear algebra problems in parallel, the user needs a high-level library where genericity, performance and portability are the main concern. The library main objectives are:

- to allow the user to work at a high level of abstraction,
- to hide many details specific to parallel programming,
- to take into account large range of machine architectures.

In addition, based on our experience on a Gaussian elimination algorithm [10], the parallelization of some routines in exact linear algebra revealed several aspects that need to be taken into account:

- **Recursion:** In parallel numerical linear algebra, routines are mainly iterative algorithms [3] with fine grain parallelism. This induces invariable block size with fixed cutting according to the matrix dimension. In these conditions, it is easier for the programmer to map and schedule tasks or threads manually.

  Nevertheless, in exact linear algebra over finite fields, algorithms such as Gaussian elimination reaches peak performances thanks to recursive sub-cubic algorithms, inducing less modular reductions [10, table 2].

  The implementation of parallel recursive algorithms using a low-level API (*e.g.*: POSIX, Windows Threads, ...) can be very hard. However, threads/tasks management can be delegated to a runtime system allowing the programmer to easily benefit from a high level of parallelism.

[*]This work is partly funded by the HPAC project of the French Agence Nationale de la Recherche (ANR 11 BS02 013).

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3599-7/15/07...$15.00
DOI: 10.1145/1235

- **Unbalanced load and communication:** In parallel computation, the role of the scheduler of a runtime system, is to assign jobs on available cores in order to optimize some criteria:

  – maximizing average workload,

  – minimizing overall completion time.

  For Gaussian elimination, in numerical linear algebra, the input matrices are mainly non singular and its principal minors are non zero. Therefore the splitting of the matrix can be done statically according to a granularity parameter. This provide a deterministic execution which is easier to parallerize. Whereas triangular decomposition over finite fields may discover rank deficiencies upon computation, thus generating tasks of unbalanced workloads. Hence, the runtime system used to parallelize such algorithms needs to provide an optimized scheduler that handles this issue.

  This problem can be better managed by clever and elaborated schedulers, especially if dataflow dependency-based task scheduling is supported. The latter is a recent programming style based on scheduling tasks that relies on computing data dependencies. Dataflow scheduling allows finer synchronizations between tasks.

- **Routines composition:** In numerical linear algebra, most parallel iterative algorithms relies on a single level of parallelism. Thus, scheduling tasks can be done manually by the programmer.

  However in parallel exact linear algebra, the execution of recursive algorithms yields to many calls of parallel routines at each recursion level. The composition of parallel routines implies different level of parallelism. By detecting data dependencies between tasks, a high level runtime system makes easier the scheduling of composed tasks. For example, task dataflow parallel programming languages leverage a runtime scheduler that is aware of dependencies between tasks.

In summary, for the parallelization of exact linear algebra libraries we want to avoid API with low level management. Instead we want to use optimized runtime systems that provide us with dataflow based synchronizations. Consequently, a high level description of parallelism is required as, for instance, OpenMP [4], TBB [16] and xKaapi [12] parallel libraries.

1.1 Motivation

The multitasking and multithreading parallelization has long existed in some manufacturers (CRAY, NEC, IBM, ...), but each had its own instructions set. The resurgence of multiprocessor machines with shared memory pushed to define a standard. A significant majority of manufacturers and builders have adopted OpenMP (Open Multi Processing) as a standard for shared memory parallelism. Many parallel libraries exists alongside OpenMP, and can be pooled in two groups:

- Annotation based API:

  – OpenMP is an API that supports multi-platform shared memory multiprocessing programming on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviour. OpenMP is still considered as standard for shared-memory architecture thanks to several advantages:

    ∗ good performance and scalability,

    ∗ mature library,

    ∗ portability,

    ∗ simple syntax, requiring little programming effort,

    ∗ allows incremental parallel implementation.

  – SMPSS - SMP SuperScalar is a task based programming environment for parallel applications based on function level parallelism. Tasks are defined with a pragma annotation right before their function definition. This annotation indicates that the following function is a task and specifies the directionality of each of the task parameters.

- Function-class based libraries

  – TBB [16] implements work stealing to balance a parallel workload across available processing cores. It is a library that implements task parallelism (fork-join) and data parallelism (parallel for).

  – CILK++ [20] is an extension to the C and C++ languages to support data and task parallelism using work-stealing policy.

  – xKaapi [12], as CILK++, implements both data and task parallelism using work-stealing policy but also with dataflow dependency between tasks.

  – StarPU is a task programming library for hybrid and heterogeneous multicore architectures.

In this work we present PALADIn that stands for Parallel Algebraic Linear Algebra Dedicated Interface which is a domain specific language dedicated to parallel exact linear algebra computation. It is included in the FFLAS-FFPACK library [7], but it can be used in any C++ linear exact algebra library. It supports OpenMP, TBB and xKaapi parallel environments and allows also the execution of the program in sequential.

A domain specific language can be implemented in C++ by using either C/C++ macros or by C++ template meta-programming. Many aspects led us to implement a macros-based language:

- By adding macros, no important modifications are to be done to the original program.

- Macros can be used also for C-based algebraic libraries.

- Simpler for the programmer and the user.

- No function call runtime overhead when using macros.

For the sake of simplicity and portability, no precompilation phase is needed to use the PALADIn library. Indeed, domain specific languages can use precompiler such as flex precompiler to generate C/C++ programs. The PALADIn library uses compilers that are installed by default in Linux distributions (g++, clang++, ...). By using g++ compiler with the released version 4.9, or Clang++ version 3.7, or earlier versions, the user can benefit from dataflow parallelism, this feature is detailed in section 2. Notes, that PALADIn can be used with older compilers, the above requirements are only needed if one wants to use dataflow parallelism.

The PALADIn library focuses on four mains aspects:

- Give an optimized parallel interface for exact linear algebra computation.
- Being able to use sequential C++ and parallel implementations using different runtime systems with a unique syntax.
- Provide the user the choice of different range cut strategies.
- Allow switching between dataflow model and explicit task synchronization with one implementation.

In section 2, we present the parallel environments supported by the PALADIn library. In section 3 we detail the PALADIn syntax and define its grammar. For each runtime library supported we show its performance on exact linear algebra routines from the FFLAS-FFPACK library in section 4. We also show that the PALADIn library has almost no overhead in performance in this section.

2. LANGAGE OF PARALLEL LIBRARIES

In this work, we use three different parallel libraries: annotation based OpenMP library and class function based xKaapi and TBB libraries. We present briefly in this section directives of each language that are used in PALADIn.

2.1 OpenMP

As shared memory machine architectures started to become prevalent, the OpenMP standard specification started in 1997 and was mainly based on loops parallelization. The concept of tasks appeared in OpenMP standard in the version 3.0 released in 2008. Thanks to these features both coarse-grained and fine-grained parallelism are possible. The latest release of OpenMP in 2013, version 4.0, adds some new features mainly support for accelerator, thread affinity and tasking extensions by adding new OpenMP clauses. In OpenMP standard different types of clauses exist to help the user set data environment management. In this work, we are interested in only few of them mostly in data sharing attribute clauses, synchronization clauses and some scheduling clauses.

OpenMP 3.0 allows two types of parallelisms:

- Parallel loops.
- Fork-join using OpenMP tasks .

In both types, data sharing attributes can be set using mainly the following clauses:

- `shared`: data within a parallel region are visible and accessible by all threads simultaneously.
- `private`: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable.
- `firstprivate`: like private except initialized to original value.
- `lastprivate`: like private except original value is updated after construct.
- `reduction`: a safe way of joining work from all threads after construct.

By default OpenMP passes all data as `firstprivate`. So, if needed, shared data can be specified by the user. One can refer to [4] for more details.

In the latest release of OpenMP 4.0 [5], dataflow parallelization model is supported via the `depend` clause, but needs g++ compiler version 4.9 or newer.

PALADIn supports OpenMP 3.x directives and the `depend` clause of OpenMP 4.0

OpenMP scheduler uses `libgomp` runtime library to handle task creation and management.

2.2 TBB

Soon after the introduction of the first multicore CPU the pentium D, Intel releases the first version of Threading Building Blocks (TBB) in 2006. TBB is a C++ template library that provides parallel algorithms and data structures avoiding to the user the need to deal with native threading. Unlike, OpenMP the library is compiler and platform independent.

The library implement task parallelism with work stealing strategy to circumvent unbalanced work load. Moreover, TBB algorithms (`parallel_for`, `parallel_reduce`, ...) are designed using fork-join tasks. Hence, every algorithm benefit from the work stealing strategy, unlike OpenMP `parallel for`. In TBB every loop based algorithm takes a functor that helps deciding the cutting strategy of the loop range.

Since version 2.1 release in 2008, TBB integrates many C++ 11 features to simplify the interface. For instance, one can easily create a task by using C++ 11 lambda function, hence avoiding the need to define specific functor.

Finally, the last feature provided by TBB is a memory allocator that takes into account many parameters to allow better scaling.

2.3 Kaapi

xKaapi stands for "Kernel for Adaptive, Asynchronous Parallel and Interactive programming". It is a C++ library that allows to execute fine/medium grain multithreaded computation with dynamic dataflow synchronizations. It is a work-stealing based parallel library that originally [12] aimed to exploit with great efficiency the computation resources of a multiprocessor cluster with a runtime support implementation. The latter is an efficient work-stealing algorithm for a macro data flow computation based on a minor extension of POSIX thread interface.

Today, the xKaapi project focuses on shared memory and CPU/GPU computation. It provides an implementation of the `libgomp` [17], the GNU implementation of the OpenMP Application Programming Interface, called `libkomp` [9]. It is an implementation of the OpenMP standard runtime based on the xKaapi library [13]. Expressing parallelism using tasks allows the programmer to choose a finer grain parallelism. But the success of such an approach depends greatly on the runtime system used.

This allows to use xKaapi tasks using an optimized runtime library `libkomp` to execute OpenMP tasks. It handles task creation and scheduling better than `libgomp` for recursive tasks [9]. Using g++4.9 or newer, `libkomp` library detects data dependencies of the `depend` clause of the OpenMP environment. This means that the PALADIn language can support dataflow parallelism using OpenMP 4.0 and xKaapi via the `libkomp` runtime library.

3. MACRO-BASED PALADIN LIBRARY

The choice of a macro based language implies some challenges:

- To give a simple interface to the user, some macros need to be overloaded. C++ allows to specify more than one definition for a function name in the same scope, which is called function overloading. When an overloaded function is called, the compiler determines the most appropriate definition to use by comparing the argument types used to call the function with the parameter types specified in the definitions. However, the compiler cannot detect overloaded macros in the preprocessing step as it would do with functions.
- To give the user more freedom, the PALADIn library allows to give a variable number of parameters. This lead us to deal with variadic macros that make it difficult to iterate over arguments, or determine the number of arguments.

3.1 Implementation examples

By using the PALAD-Interface, the user can benefit from data or task parallelism. We do not intend to explain the PALADIn keywords here. Further explanation on the grammar and description are given in sections 3.2 to 3.5. In this section, we only give two code examples using PALADIn to illustrate its syntax. The first example show the parallel loop syntax, and the second example illustrate the task syntax.

- **Parallel loop:**
  Let us consider three arrays `T`, `T1` and `T2`. The arrays can be any C/C++ structure. In this example, we sum `T1` with `T2` and store the result in `T` componentwise. The simple C++ code doing this operation is:

  Listing 1: Loop summing of two arrays in C++

  ```
  for(size_t i = 0 ; i < n ; ++i){
    T[i] = T1[i] + T2[i];
  }
  ```

  The translation of the above code in the PALALDIn syntax is:

  Listing 2: Loop summing of two arrays with PALADIn

  ```
  PARFOR1D(it, 0, n, SPLITTER,
    T[it.begin()] = T1[it.begin()]+T2[it.
        begin()];);
  ```

  Using PALADIn parallel for, the user can set the variable `SPLITTER` to specify the desired strategy to cut chunks of the loop range iterated with `it`. More details on setting this variable and other variables can be found in the PALADIn description in sections 3.4 and 3.6.

- **Task parallelism:**
  In this example we illustrate the PALADIn task syntax. Let us consider a free function `apxy` that uses three parameters `a`, `x` and `y` and computes `y+= ax`.

  Listing 3: Task call with PALADIn

  ```
  void axpy(const Element a, const Element b
      , Element y){
    y += a*x;
  }

  TASK(MODE(READ(a,x) READWRITE(y)),
    apxy(a,x,y));
  ```

  The `READ` macro specifies that the arguments `a` and `x` are only read in the task execution. The `READWRITE` macro indicates that the variable `y` is in read and write mode during the task execution.

3.2 PALADIn description

The PALADIn extends the C++ language with new keywords that enable two complementary parallel programming paradigms:

- Data parallelism (*i.e.* SPMD *Single Program Multiple Data*), thanks to parallel regions defined by the `PARFOR1D` and `PARFOR2D` keywords.
- Task parallelism:
  - serial-parallel computations (*i.e.* fork-join) with `PAR_BLOCK` and `SYNCH_GROUP` keywords;
  - asynchronous task parallelism (*i.e.* tasks which synchronizations are defined by data dependency instead) inside the `TASK` keyword with `READ`, `WRITE` and `READWRITE` keywords and also between dependent tasks by `CHECK_DEPENDENCIES` keyword.

To enable parallelism in the main sequential stream of instructions, `PARFOR1D(`$i, f, l, splitter, I$`)` declares a new parallel loop where the variable i ranges the interval $[f, l[$ to execute the body I. At each step, the interval is split (eventually recursively) thanks to splitter method in sub-intervals that are concurrently computed. The `PARFOR1D` is terminated when all sub intervals are computed. Note that, like in conventional SPMD programming, I may contain branching according to the current iteration value i. The `PAR_BLOCK{`I`}` is the special case where the interval contains only one element.

`PARFOR1D`, `PARFOR2D` or `PAR_BLOCK` define a new parallel region. Like in OpenMP, a parallel region shall not be nested within another parallel region as it can cause oversubscription, (i.e., the number of busy threads is greater than the number of cores), which may degrade the speedup. This over-decomposition of the workload leads to additional unnecessary context switches with non negligible cost.

`SYNCH_GROUP(`I`)` – with I denoting a block of instructions – enables to declare a new synchronization point (*i.e.* local barrier) : at execution, the `SYNCH_GROUP(`I`)` instruction is passed only after completion of all parallel computations forked by I.

Indeed, within a `SYNCH_GROUP`, the instruction `TASK(`D, I`)` forks the execution of the instruction I. The default synchronization (local barrier) after I is at the end of the

`SYNCH_GROUP`. Moreover, D defines additional synchronizations from expressing dataflow dependencies; indeed D optionally defines the access mode to objects through four lists of variables:

- `REFERENCE`(*variables list*) : those variables are passed by reference to I (by default, variables are passed by value, similarly to mod `firstprivate` in OpenMP);
- `READ`(*variables list*) : those variables are read by I, but not modified;
- `WRITE`(*variables list*) : those variables are written, but not read;
- `READWRITE`(*variables list*) : those variables are read then written (update).

Note that the `MODE` macro allows to describes any task dependencies that form a directed acyclic graph. Only those explicit dependencies define synchronizations within a group, before the local barrier at the end of the group.

3.3 PALADIn grammar

PALADIn extends the instruction set of `C++` with new instructions; the following grammar defines the sequences of those instructions (traces) that are considered not only valid syntactically but also at execution. In particular, it doesn't allow not only incorrect syntax constructions but also incorrect in term of the performance of executions, as for instance preventing nesting of `PARFOR` or `PAR_BLOCK`.

Any trace (full instructions stream) of a valid PALADIn program is an instance of PALADIN_INSTR. In this grammar, SEQ_INSTR denotes any sequential instruction (at any level of trace) resulting from the execution of a standard `C++` block of instructions, excluding the new PALADIn keywords.

We extend this grammar by adding new set of instructions (lexicographic units are in bold):

PALADIN_INSTR ⟶ SEQ_INSTR
| `PAR_BLOCK`{SYNCH_INSTR}
| `PARFOR1D`(INTERAVL1D, SYNCH_INSTR)
| `PARFOR2D`(INTERVAL2D, SYNCH_INSTR)
| (PALADIN_INSTR;)*

SYNCH_INSTR ⟶ SEQ_INSTR
| `SYNCH_GROUP`(ASYNCH_INSTR)
| (SYNCH_INSTR;)*

ASYNCH_INSTR ⟶ SYNCH_INSTR
| `TASK`(DEPENDENCIES, ASYNCH_INSTR)
| `FOR1D`(INTERVAL1D, ASYNCH_INSTR)
| `FOR2D`(INTERVAL2D, ASYNCH_INSTR)
| (ASYNCH-INSTR;)*

DEPENDENCIES ⟶ MODE((CONSTREF_STATE)?
| (REF_STATE)?
| (READ_STATE)?
| (WRITE_STATE)?
| (READWRITE_STATE)?)

INTERVAL1D ⟶ IDF, INT_EXPR, INT_EXPR, SPLITTER

INTERVAL2D ⟶ IDF, INT_EXPR, INT_EXPR,
IDF, INT_EXPR, INT_EXPR, SPLITTER

CONSTREF_STATE ⟶ ϵ | `CONSTREFERENCE`(VAR)+

REF_STATE ⟶ ϵ | `REFERENCE`(VAR)+

READ_STATE ⟶ ϵ | `READ`(VAR)+

WRITE_STATE ⟶ ϵ | `WRITE`(VAR)+

READWRITE_STATE ⟶ ϵ | `READWRITE`(VAR)+

SPLITTER ⟶ INT_EXPR, STRATEGY

STRATEGY ⟶ `SINGLE`
| `ROW_FIXED`
| `COLUMN_FIXED`
| `BLOCK_FIXED`
| `ROW_THREADS`
| `COLUMN_THREADS`
| `BLOCK_THREADS`
| `GRAIN_SIZE`
| `TWO_D`
| `THREE_D_INPLACE`
| `THREE_D_ADAPT`
| `TWO_D_ADAPT`
| `THREE_D`

3.4 Cutting strategies

The *SPLITTER* parameter in the previous grammar gives the range cut strategy used to execute the corresponding program inside the loop. We present here all the cutting strategies that are used in the following macros: `PARFOR1D` `PARFOR2D`, `FOR1D` and `FOR2D`.

PALADIn implements an overall of 8 different matrix cutting strategies for iterative algorithms that are grouped in two categories:

- One dimension cutting strategies:

  `ROW_THREADS` This cutting strategy take into account the number of processors p, and splits the rows of the matrix into exactly p row slabs.

  `ROW_FIXED` This cutting strategy cuts the rows of the matrix with a fixed grain size set by default to 256.

  `COLUMN_THREADS` As the `ROW_THREADS` strategy, `COLUMN_THREADS` take into account the number of processors p but splits the columns of the matrix into exactly p column slabs.

  `COLUMN_FIXED` This cutting strategy cuts the columns of the matrix with a fixed grain size set by default to 256.

- The two dimensions cutting strategies:

  `BLOCK_THREADS` This strategy cuts the two dimensions of the output matrix. When performing the operation $C \leftarrow A \times B$, it splits A in s row slabs and B in t column slabs, and splits the matrix C in $s \times t$ tiles. The values for s and t are chosen such that their product equals the number of threads available.

  `BLOCK_FIXED` This strategy cuts the two dimensions of the matrix C as the `BLOCK_THREADS` cutting strategy by with a fixed grain size set to 256. This gives tiles of size 256×256.

`GRAIN_SIZE` This strategy allows the user to give a block size b. Thus tiles are of size $b \times b$.

The `SINGLE` strategy is the strategy that does not cut the matrices and thus allows a standard sequential behavior of the loop.

In the case of recursive matrix multiplication algorithms that involve splitting of three dimensions, one can use 5 different recursive cuttings provided by PALADIn. We show here these cutting strategies, that are dedicated for the parallel general matrix multiplication (`pfgemm`) operation: computing $C \leftarrow \alpha A \times B + \beta C$, where A, B and C are dense matrices with dimensions respectively (m, k), (k, n) and (m, n).

`TWO_D` The 2D recursive partitioning performs a 2×2 splitting of the matrix C at each level of recursion. Each recursive call is then allocated a quarter of the number of threads available. This constrains the total number of tasks created to be a power of 4 and the splitting will work best when the number of threads is also a power of 4.

`TWO_D_ADAPT` The 2D recursive adaptive partitioning cuts the largest dimension between m and n, at each level of recursion, creating two independent recursive calls. The number of threads is then divided by two and allocated for each separate call (with a discrepancy of allocated threads of at most one). This splitting better adapts to an arbitrary number of threads provided.

`THREE_D_INPLACE` The 3D in-place recursive cutting strategy performs 4 multiply calls, waits until blocks elements are computed and then performs 4 multiply and accumulation. This variant is called *inplace* since blocks of matrix C are computed in place.

`THREE_D` performs 8 multiply calls in parallel and then performs the add at the end. To perform 8 multiplications in parallel we need to store the block results of 4 multiplications in temporary matrices. As in the previous routine, each task calls recursively the routine.

`THREE_D_ADAPT` The 3D recursive adaptive cutting strategy cuts the largest of the three dimensions in halves. When the dimension k is split, a temporary is allocated to perform the two products in parallel. As the split the k dimension introduces some overhead, one can introduce a weighted penalty system to only split this dimension when it is largely greater than the other dimensions: with a penatly factor of p, the dimension k is split only when $\max(m, n) < pk$.

3.5 Implementation issues and extensions

The PALADIn language is implemented by macro definitions with implementations provided for sequential C++ programs and several target parallel environments. Currently for the C++ language the libraries OpenMP, TBB and xKaapi are targeted. Thus syntax of C++ is not modified, enabling to use PALADIn for any program written in C++.

While OpenMP 4 and xKaapi supports data dependencies, environments like OpenMP 3 or TBB does not. Hence, to guarantee synchronizations related to data dependencies in a language with no support of it, we have defined a new instruction `CHECK_DEPENDENCIES` that forces the dependencies previously defined in the current group. This implementation may be pessimistic but ensures PALADIn independence from the underlying parallel environement. In our OpenMP 4 and xKaapi implementations, since dependencies are ensured at task creation, `CHECK_DEPENDENCIES` has no effect. But in OpenMP 3 and TBB it is compiled as a local synchronization barrier within the group.

3.6 Code examples

We show here the PALADIn semantics and its equivalence in OpenMP and TBB on the axpy example given in section 3.1. The task that performs this operation is invoked by :

```
void axpy(const Element a, const Element b,
    Element y){
  y += a*x;
}

SYNCH_GROUP(
  TASK( MODE( READ(x, y) READWRITE(y)),
    axpy(a, x, y)););
```

We show its equivalent implementation with OpenMP 3 syntax:

```
#pragma omp task
  axpy(a, x, y);
#pragma omp taskwait
```

With OpenMP 4 syntax using the "depend" clause:

```
#pragma omp task depend(in:a,x) depend(inout:y)
  axpy(a, x, y);
#pragma omp taskwait
```

Using lambda function, the syntax with tbb becomes:

```
tbb::task_group g;
g.run([&y, a, x](){axpy(a, x, y);});
g.wait();
```

The `SYNCH_GROUP` macro ensures that a local synchronization is set at the end of the AXPY task.

Below we illustrate two examples using the PALADIn syntax, and show, in section 4, how the cutting strategy can have an impact on the parallel performance.

The first example depicts three different implementations to write a parallel loop of a C++ program: one using the OpenMP parallel loop syntax, Listing 4, the other one using TBB `parallel_for`, Listing 5, and the PALADIn syntax, Listing 6, for the parallelization of the same loop by using different cutting strategies. In this example we attempt to perform the operation $C \leftarrow A + B$, where the matrices A, B and C are stored in a row major manner. The `pfadd` routine processes this operation on several pairs of operands simultaneously which allows each thread to execute a vectorized add operation.

Listing 4: *parallel fadd* with OpenMP parallel loop

```
void pfadd(const Field & F, const Element *A,
    const Element *B, Element *C, size_t n){
  #pragma omp parallel for
  for(size_t i = 0 ; i < n ; ++i){
    FFLAS::fadd(F, 1, n, A+i*n, n, B+i*n, n, C+
        i*n);
  }
}
```

Listing 5: *parallel fadd* with TBB

```
void pfadd(const Field & F, const Element *A,
    const Element *B, Element *C, size_t n){
  parallel_for(blocked_range<size_t>(0, n),
    [&](blocked_range<size_t> & r){
      for(size_t i = r.begin() ; i < r.end() ;
          ++i){
        FFLAS::fadd(F, 1, n, A+i*n, n, B+i*n, n
            , C+i*n, n);
      }
    });
}
```

Listing 6: *parallel fadd* with PALADin

```
void pfadd(const Field & F, const Element *A,
    const Element *B, Element *C,
size_t n){
 FFLAS::ParSeqHelper::Parallel splitter = FFLAS
     ::ParSeqHelper::Parallel(32, ROW_THREADS);
  PARFOR1D(it, 0, n, splitter,
    FFLAS::fadd(F, it.end()-it.begin(), n, A+it
        .begin()*n, n, B+it.begin()*n, n, C+it.
        begin()*n, n););
}
```

The `splitter` parameter, in Listing 6, can be defined as a parallel or a sequential helper. In the sequential case, no cutting strategy will be used. In this example it is set as a parallel helper with `FFLAS::ParSeqHelper::Parallel` and takes two arguments to specify a cutting strategy: the number of threads (i.e. 32 threads in this example) and the strategy of splitting (for instance the `ROW_THREADS` strategy) of the matrix.

As a second example, we use the sparse matrix-vector product over a finite field. In this operation, the matrix is stored in the classical Compress Sparse Rows (CSR) format [8], see Figure 2. The CSR format is composed of 3 arrays: the first one to store the value of non zeros, the second one to store the column indices of the non zeros elements, and a third one containing pointer of where the ith rows start in the two previous arrays. Hence the CSR save some memory which increase performance as the SpMV operation is memory bound. The OpenMP implementation is shown in Listing 7, the TBB implementation in Listing 8 and the PALADin implementation in Listing 9. The performance behaviour of this operation and its implementations are explained in subsection 4.1.

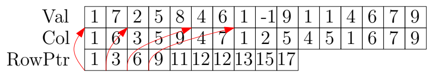
We illustrate, in this example, the implementation of a simple parallel loop with OpenMP and TBB on a sparse matrix-vector multiplication operation.

Listing 7: Parallel implementation of SpMV with OpenMP

```
void spmv(const Field & F, const CSRMat & A,
    const Element *x, Element *y){
  #pragma omp parallel for
  for(size_t i = 0 ; i < n ; ++i){
    size_t start = M.rowPtr[i], stop = M.rowPtr
        [i+1];
    for(size_t j = start ; j < stop ; ++j){
      y[i] += M.val[j] * x[M.col[j]];
    }
  }
}
```

$$M = \begin{bmatrix} 1 & & & & & 7 & & & \\ & & 2 & & 5 & & & & 8 \\ & & & 4 & & & 6 & & \\ 1 & -1 & & & & & & & \\ & & & & 9 & & & & \\ & & & & & & & & \\ & & & 1 & 1 & & & & \\ 4 & & & & & 6 & & & \\ & & & & & & 7 & & 9 \end{bmatrix} \tag{1}$$

Figure 1: A matrix M.



Val	1	7	2	5	8	4	6	1	-1	9	1	1	4	6	7	9
Col	1	6	3	5	9	4	7	1	2	5	4	5	1	6	7	9
RowPtr	1	3	6	9	11	12	12	13	15	17						

Figure 2: CSR storage of the matrix M.

Listing 8: Parallel implementation of SpMV with TBB

```
void spmv(const Field & F, const CSRMat & A,
    const Element *x, Element *y){
  parallel_for(blocked_range<size_t>(0, A.m),
    [&](blocked_range<size_t> & r){
      for(size_t i = r.begin() ; i < r.end() ;
          ++i){
        size_t start = M.rowPtr[i], stop = M.
            rowPtr[i+1];
        for(size_t j = start ; j < stop ; ++j){
          y[i] += M.val[j] * x[M.col[j]];
        }
      }
    });
}
```

Listing 9: Parallel implementation of SpMV with PALADin

```
void spmv(const Field & F, const CSRMat & A,
    const Element *x, Element *y){
  PARFOR1D(it, 0, A.m, splitter,
      for(size_t i = it.begin() ; i < it.end()
          ; ++i){
        size_t start = M.rowPtr[i], stop = M.
            rowPtr[i+1];
        for(size_t j = start ; j < stop ; ++j){
          y[i] += M.val[j] * x[M.col[j]];
        }
      }
}
```

4. PERFORMANCE OF PALADIN LIBRARY

In this section we show the performance the PALADIn library for the data parallelism and task parallelism programming styles. Experiments are done on 32 cores (4 NUMA nodes with 8 cores each) Xeon E4620 2.2Ghz.

4.1 parallel loop performance

Parallelizing loops with OpenMP can be very simple using `#pragma omp parallel for`. This lets the scheduler of OpenMP to choose the default mode for cutting loop iterations in chunks and distribute them on available resources. The user can set the strategy for the scheduler to specify the size of chunks that can be executed statically or dynamically. Using the PALADIn cutting strategies one can have better performance without important modification of the program.

We show here the performance of the two examples described in the previous section.

Matrix Addition.

Table 1 shows the performance of Listing 4, Listing 5 and Listing 6 described before. For the PALADIn implementation two cutting strategies are used, `ROW_THREADS` and `ROW_FIXED`, according to one dimension to show that for a simple parallel loop one can achieve better performance using the PALADIn cutting strategy than the default cutting strategies given by a standard parallel model such as OpenMP and TBB. We execute the PALADIn cutting strategies with OpenMP and TBB. Table 1 demonstrates that the best performance are obtained with the `ROW_THREADS` cutting strategy for the parallel dense matrix-matrix addition operation. Even when using the same cutting strategy (`ROW_THREADS`) we can see that TBB is slower than OpenMP. Since the executions are done on 32 cores of a NUMA machine architecture, TBB does not chose obviously the best cutting strategy adapted to the machine hierarchy.

Matrix dimension	1000	2000	3000	4000
omp parallel for	1.7	4.2	8.4	15.0
omp `PARFOR1D(ROW_THREADS)`	**1.2**	**3.6**	**8.2**	**14.0**
omp `PARFOR1D(ROW_FIXED)`	1.9	5.8	9.8	17.0
TBB parallel_for	5.0	16.0	28.0	160.0
TBB `PARFOR1D(ROW_THREADS)`	**1.4**	**6.6**	**15.0**	**30.0**
TBB `PARFOR1D(ROW_FIXED)`	2.6	11.0	23.0	34.0

Table 1: Timings in milliseconds of the PALADIn language using two different cutting strategies compared to openmp "parallel for" for the fadd operation of two square matrices.

SpMV Operation.

For the experiments we used two matrices:

- ffs619 of dimensions 653365×653365 with an average of 100 non zeros elements by row
- ffs619 of dimensions 3602667×3602667 with an average of 110 non zeros elements by row

The computation is done over the finite field $\mathbb{Z}/524309\mathbb{Z}$, using 8 cores, results are reported in table 2.

The non zeros elements of the matrices are not uniformly distribute, more than 90 % of the non zeros elements are in the first thousand rows and the last rows have at most 3 elements.

The OpenMP implementation does not perform well because the scheduling strategy cannot deal with unbalanced workload provide by the particular distribution of the non zeros elements. The TBB implementation perform better because TBB uses task parallelism with work stealing strategy to compute the loop, thus balancing the workload more efficiently over the cores. However, by default the TBB task are composed of at most two rows which mean that for the sparsest part of the matrix, a task only compute 8 multiplications and 8 additions. Hence, the overhead of TBB task management greatly impact performances. With the PALADIn implementation, the `ROW_THREADS` cutting strategy produces only 8 tasks where each one as a part of the loop range, this strategy is similar to the OpenMP `parallel for`, thus produces bad performances. The `ROW_FIXED` cutting strategy split the loop in a fixed number of iterations (256 in this benchmarks) allowing the scheduler to efficiently balance the workload over the cores and the tasks are big enough to cover the management overhead.

Matrix	ffs619	ff809
OpenMP	0.49	0.26
omp `FOR1D(ROW_THREADS)`	0.40	0.24
omp `FOR1D(ROW_FIXED)`	**2.00**	**0.95**
TBB	0.95	0.43
tbb `FOR1D(ROW_THREADS)`	0.44	0.26
tbb `FOR1D(ROW_FIXED)`	**1.99**	**0.90**

Table 2: Performance in Gfops of PALADIn compared to OpenMP and TBB "parallel for" for the CSR spmv operation of two sparses matrices arising in the discrete logarithm problem [2].

We can see clearly, in this table, that using the cutting strategy `ROW_FIXED` one can achieve at least a speed-up of 2 to perform a sparse matrix-vector multiplication operation.

4.2 Fork-join performance

In this section we show performance of PALADIn interface implementing exact linear algebra routines using tasks. We compare here execution speed of different cutting strategies, described in section 3, for the matrix product operation using OpenMP (using libgomp runtime library) and xKaapi (using `libkomp` runtime library).

In these experiments, we use the effective Gfops metric: $Gfops = \frac{\text{\# of field ops using classic matrix product}}{\text{time}}$. It stands for Giga field operations per second and is $\frac{2mnk}{\text{time}}$ for the product of an $m \times k$ by a $k \times n$ matrix, and $\frac{2n^3}{3\text{time}}$ for the Gaussian elimination of a full rank $n \times n$ matrix.

Experiments are conducted on square matrices with dimensions between 1000 and 15000 and elements are over the finite field $\mathbb{Z}/131071\mathbb{Z}$, using 32 cores.

Figures 4 and 5, for sake of simplicity, show the behavior of the best 5 different cutting strategies for the parallel matrix multiplication operation. With the `libgomp` runtime, the `BLOCK_THREADS` cutting strategy is much faster, as recursive tasks seem to be poorly handled. Thanks to its efficient management of recursive tasks, the `libkomp` runtime behaves better for the recursive variants. Using TBB tasks, all cutting strategies have almost the same behavior when matrix dimensions gets bigger. These experiments demonstrates also that TBB handles better than OpenMP parallel recursive tasks and this shows the impact of a work-stealing based library on the overall performance.

4.3 Dataflow dependencies performance

In this section, we show the performance of PALADIn library when the dataflow parallelization is set.

Figure 6 shows the behavior of an iterative algorithm that computes the PLUQ decomposition [11] with explicit synchronizations (red and blue curves) and with data dependency synchronizations (black and yellow curves). In this

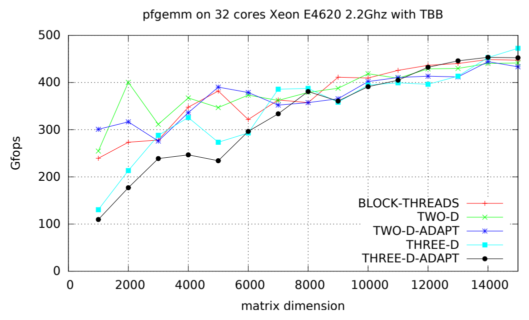


Figure 3: Speed of different matrix multiplication cutting strategies using TBB tasks

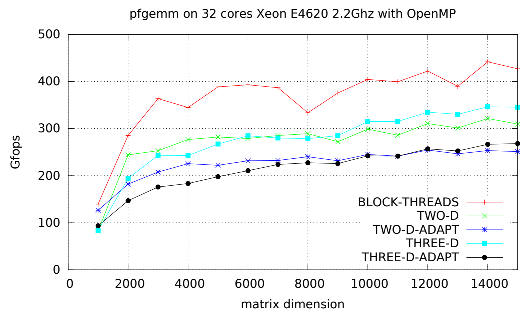


Figure 4: Speed of different matrix multiplication cutting strategies using OpenMP tasks

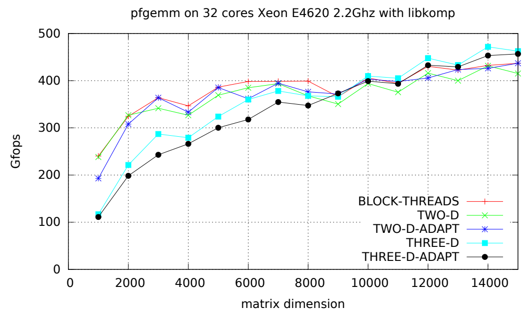


Figure 5: Speed of different matrix multiplication cutting strategies using xKaapi tasks

experiment we used an iterative version of the algorithm described in [11]. Since TBB does not support dataflow parallelization, the execution speed of the algorithm is shown using the two runtime systems OpenMP and xKaapi.

This figure demonstrates that an algorithm that generates many dependent tasks could take advantage from the dataflow parallelization model supported in the PALADIn library.

5. CONCLUSION

We presented in this work, the PALADIn interface that allows the user, using mainly C macros, to write C++ code and benefit from sequential and parallel executions on shared memory architectures. We have shown three parallel environment libraries: OpenMP, TBB and xKaapi, that are

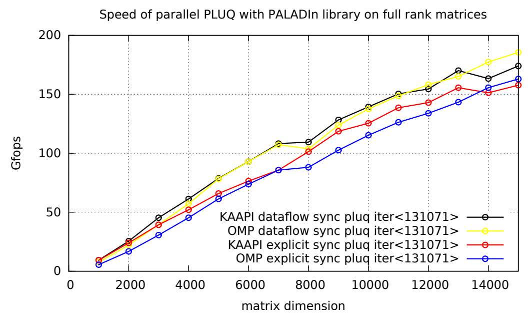


Figure 6: Speed of parallel PLUQ decomposition using the PALADIn library with and without dataflow dependencies between tasks

supported in this domain specific language. This interface provides data parallelism and task parallelization. Hence, depending on the runtime system used, the task parallelization can be performed either by using explicit synchronizations or using data-dependency based synchronizations.

We have proved that, comparing to OpenMP and TBB parallel for, the diversity of matrix cutting strategies provided in this language, helps the user to obtain always better performance.

The PALADIn interface can be used in any C++ library for linear algebra computation and gets the best parallel performance from three supported runtime systems.

Further extensions of the PALADIn library can be implemented, especially when detecting dependencies between tasks. For now, data dependencies are detected thanks to the pointer passed in parameters. The computation of data dependencies could be affected and the result could be incorrect when treating with overlapping blocks. In the latter case, the range of the blocks can be passed in parameter in the macros `READ`, `WRITE` and `READWRITE`.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammarling, J. Demmel, C. H. Bischof, and D. C. Sorensen. “LAPACK: a portable linear algebra library for high-performance computers”. In: *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*. 1990, pp. 2–11.

[2] R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. “Discrete Logarithm in GF(2809) with FFS”. In: *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*. 2014, pp. 221–238.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[4] O. A. R. Board. *OpenMP Application Program Interface version 3*. 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[5] O. A. R. Board. *OpenMP Application Program Interface version 4*. 2013. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[6] W. Bosma, J. J. Cannon, and C. Playoust. "The Magma Algebra System I: The User Language". In: *J. Symb. Comput.* 24.3/4 (1997), pp. 235–265.

[7] B. Boyer, A. Breust, J.-G. Dumas, P. Giorgi, C. Pernet, Z. Sultan, and B. Vialla. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*. v2.0.0. http://linalg.org/projects/fflas-ffpack. 2014.

[8] B. Boyer, J. Dumas, and P. Giorgi. "Exact sparse matrix-vector multiplication on GPU's and multicore architectures". In: *Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, July 21-23, 2010, Grenoble, France*. 2010, pp. 80–88.

[9] F. Broquedis, T. Gautier, and V. Danjean. "libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms". In: *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*. 2012, pp. 102–115.

[10] J. Dumas, T. Gautier, C. Pernet, and Z. Sultan. "Parallel Computation of Echelon Forms". In: *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. 2014, pp. 499–510. DOI: 10.1007/978-3-319-09873-9_42.

[11] J. Dumas, C. Pernet, and Z. Sultan. "Simultaneous computation of the row and column rank profiles". In: *International Symposium on Symbolic and Algebraic Computation, ISSAC'13, Boston, MA, USA, June 26-29, 2013*. 2013, pp. 181–188.

[12] T. Gautier, X. Besseron, and L. Pigeon. "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors". In: *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*. 2007, pp. 15–23.

[13] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. "XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures". In: *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. 2013, pp. 1299–1308.

[14] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. Version 2.4.0, http://flintlib.org. 2013.

[15] *Intel Math Kernel Library. Reference Manual*. ISBN 630813-054US. Santa Clara, USA: Intel Corporation, 2009.

[16] *Intel Threading Building Blocks*. Santa Clara, USA: Intel Corporation, 2008. URL: https://www.threadingbuildingblocks.org/.

[17] J. Jelinek and *et al. The GNU OpenMP implementation*. 2014. URL: https://gcc.gnu.org/onlinedocs/libgomp.pdf.

[18] M. library. *http://icl.cs.utk.edu/magma/index.html*.

[19] P. library. *http://icl.cs.utk.edu/plasma/index.html*.

[20] K. H. Randall. "Cilk: Efficient Multithreaded Computing". PhD thesis. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.