# Program Sequentially, Carefully, and Benefit from Compiler Advances for Parallel Heterogeneous Computing

Mehdi Amini, Corinne Ancourt, Béatrice Creusillet, François Irigoin, Ronan Keryell

# Program Sequentially, Carefully, and Benefit from Compiler Advances for Parallel Heterogeneous Computing

Mehdi Amini[1,2] Corinne Ancourt[1] Béatrice Creusillet[2]
François Irigoin[1]
Ronan Keryell[2]

—

[1]MINES ParisTech/CRI, *firstname.lastname*@mines-paristech.fr
[2]SILKAN, *firstname.lastname*@silkan.com

September 26th 2012

**Abstract**

The current microarchitecture trend leads toward heterogeneity. This evolution is driven by the end of Moore's law and the frequency wall due to the power wall. Moreover, with the spreading of smartphone, some constraints from the mobile world drive the design of most new architectures. An immediate consequence is that an application has to be executable on various targets.

Porting and maintaining multiple versions of the code base requires different skills and the efforts required in the process as well as the increased complexity in debugging and testing are time consuming, thus expensive.

Some solutions based on compilers emerge. They are based either on directives added to C like in OpenHMPP or OpenACC or on automatic solution like PoCC, Pluto, PPCG, or Par4All. However compilers cannot retarget in an efficient way any program written in a low-level language such as unconstrained C. Programmers should follow good practices when writing code so that compilers have more room to perform the transformations required for efficient execution on heterogeneous targets.

This chapter explores the impact of different patterns used by programmers, and defines a set of good practices allowing a compiler to generate efficient code.

# Contents

**Keywords:** parallel programming, automatic parallelization, coding rules.

# 1 Introduction

As mono-core processor technology has reached its limits, and as our computing needs are growing exponentially, we are facing an explosion of proposals in the current architectural trends, with more arithmetic units usable at the same time, vector units to apply similar operations on the elements of short vectors (such as SSE, AVX, NEON...), several computing cores in the same processor die, several processors sharing or not the same memory, up to more heterogeneous architectures with hardware accelerators of specific functions, FPGA or massively parallel GPU.

As a consequence, an application has to be executable on various platforms, at a given time as well as during its whole life cycle. This execution has to meet some efficiency criteria, such as energy or computational efficiency, that may not be compatible. Thus the development and maintenance costs become crucial issues, and automatic optimization and parallelization techniques, whereas often considered as unrealistic silver bullets, are coming

back to the front stage as a partial solution to lower the development and maintenance cost.

Of course, automatic parallelization cannot be expected to produce miracles, as the problem is intractable in the general case. Reaching the highest possible performance for a given application on a specific architecture will always require some kind of human intervention, ending up sometimes with a complete rewrite from scratch.

However, and fortunately, sufficiently *well-written* sequential programs can expose enough parallelism that automatic parallelizers can detect, and for which they can do a sufficiently good job. In particular, we advocate that, before any attempt at manual or automatic parallelization, programs should avoid sequential processor optimizations and stick as much as possible to the high-level semantics of the algorithms. Thus, instead of writing code for a specific class of architectures, programmers and more widely code generators must learn how to write sequential programs, efficiently targeting the current class of advanced compilers, namely compilers with loop optimizers, vectorizers and automatic parallelizers.

This document, based on the long experience of its authors in this area, mainly on the PIPS [20, 1] and PAR4ALL [2, 34] projects, details some good programming practices for imperative languages that are compliant with the current capabilities of automatic parallelizers, and more generally optimizers, to maximize the precision of analyses and thus help the tools in generating efficient code. As a preliminary clue, and not surprisingly, let us say that the more structured a program is, the easier it is to analyze.

Since parallelism is present at many levels in current architectures, even a program executed on a single core can benefit from being well written, for example by allowing the use of vector instruction sets or computing some parts at compile-time through partial evaluation.

In the remainder of this document, we focus mainly on the C language, and we assume that the reader is knowledgeable in the C99 programming language[1]. However, the principles can be generalized for other languages.

In this chapter, we give in Section 2 some advice on program structure and control flow. The Section 3 presents some points related to data structures and types. Then the Section 4 introduces the concept of directives and pragmas to help the compiler in its quest to efficient parallel code. Section 5 recalls the interest of already parallelized libraries. Before concluding, Section 6 details some constraints related to object-oriented languages.

This work has been done within the OpenGPU project, where the authors have been involved in developing compilers for GPU, mainly related to C and DSL to OpenCL translation.

[1] And more specifically the ISO/IEC 9899:TC3 WG14 standard the last public draft can be found here: `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf`. We could focus on C11, but this latest version is too recent to be widespread yet.

```
int kernel(int n, int m) {
  int a[n][m], b[n][m];
#pragma acc kernels
  {
    int i=0;
    while(i<n) {
      for(int j=0; j<m; j++) {
        a[i][j]=i+j;
        if(i==j)
          a[i][j]=i-j;
        b[i][j]=a[i][j];
      }
      i++;
    }
  }
}
```

Figure 1: OpenACC example with a while loop. The PGI compiler is able to parallelize this loop nest and schedule it on GPU.

# 2 Program Structure and Control Flow

Most optimizers rely on the good properties of the program control flow to generate efficient results. Thereby, a program with poorly structured computational parts (which contains `goto`, `break`, `continue`, `exit()`, multiple returns...) prevent further code transformations and automatic optimizations. Of course some tools such as PIPS [20, 1] or the PGI Compiler have some restructuring capabilities, as shown in the listing from Figure 1, but they have their limitations. So a good practice is to adopt a well-structured programming style, or at least to restrict the use of poorly structured code to inherently sequential parts. More generally well-structured programming has to be used in parallel parts, and if not, other way to express parallelism has to be used, for example by relying on hand-coded libraries as shown in Section 5 or by hiding messy code as in Section 2.3.

## 2.1 Well-Formed Loops...

Many advanced optimization techniques, such as automatic parallelization, traditionally focus on loops — which are usually the most intensive parts of the programs — and more specifically on `for` loops with good properties, namely loops which are similar to Fortran `do` loops: the loop index is an integer, initialized before the first iteration, and its value is incremented by a constant value and compared to loop-invariant values at each iteration, and not changed by the loop body itself. The number of iterations is known before the loop is entered. Note that language extensions such as OpenMP [28], OpenACC [26] and

HMPP [8] provide parallel constructs for `do` and such *well-formed* `for` loops, but not for general `for` or `while` loops.

Tools based on the Polyhedral model [12, 22, 4, 30, 9] have long been historically very strict, by also enforcing that the initial value, the lower and the upper bounds be affine integer expressions of the surrounding loop indices and loop-invariant values, as well as `if` conditions and array indices. This is also a requirement of several compilers such as Cetus or the R-Stream compiler [33].

The polyhedral model has been extended to lift some of these conditions, and these extensions are available in LooPo [15] and PoCC [6, 30]. It must also be noted that applying a series of pre-processing transformations – such as constant propagation or induction variable substitution — can raise the number of good candidates [5]. However, this is quite undesirable in source-to-source compilers.

So, even when these conditions are not compulsory, they are greatly encouraged, as most tools take advantage of them to detect parallelism and/or generate more efficient code. For instance, the following *manually optimized* piece of code

```
double a[n][m][l];
double * p = &a[0][0][0];
for(int i = 0; i < n*m*l; i++)
  p[i] = ...;
```

can be parallelized by Par4All OpenMP compiler [34], but the communications cannot be generated accurately in the GPU version because of the non-linear loop upper bound.

In most tools [29, 9, 32, 31, 33], loop transformations only target *well-formed* `for` loops, while other compilers also deal with `while` loops [15, 6]. In between, PIPS tries to detect `for` and `while` loops which are equivalent to *well-formed* loops. For that purpose, it is important to write these loops in such a way that they can be detected as Fortran-like `do` loops. In particular, this implies using simple comparison and increment expressions, and avoiding putting the `for` body code inside the increment expression as in:

```
for (i=0; i<n; a[i] = i, i++);
```

Also, incrementing an integer index should be preferred to directly incrementing a pointer:

```
for (p=a; p<a+n; p++)
   *p = ...;
```

should be replaced by:

```
for (int i = 0; i<n; i++)
   a[i] = ...;
p = &a[n]; // if the value of p is required later
```

## 2.2 ... and Loop Bodies

The conditions on the loop bodies characteristics depend on the tools capabilities. For instance the PGI Compiler used to not parallelize loops containing `if` conditions [36] in

previous versions. Tools relying on fully polyhedral techniques historically require that internal `if` conditions be affine integer expressions [12, 22], but this restriction has been recently lifted [6, 30]. The R-Stream compiler lies somewhere in between, allowing some non-affine condition expressions. PIPS allows any internal control flow by using approximation techniques [10], provided that there is no jumping construct from and to the loop body. However, it must be kept in mind that, in most cases, the more information is available to the optimizer, the more efficient it can be[2].

Most tools offer no or very limited interprocedural capabilities. For instance, ROSE and PGI Compilers [32, 29] do not parallelize loops with inner function calls, but this can be by-passed in PGI compiler with the `-Mconcur=cncall` option that specifies that all loops containing calls can be safely parallelized, or by activating automatic *inlining*. The R-Stream compiler [33] assumes by default that functions have no side-effects, which may lead to incorrect code; but it provides a mechanism — called *image function* — for the user to describe the side-effects of library functions using stub functions.

On the contrary, PIPS has been designed from the very beginning as an interprocedural parallelizer [37, 20], and uses interprocedural summarizing techniques to take into account memory accesses hidden in function calls. Cetus [31] also has some simple interprocedural capabilities, which allows to parallelize some loops with function calls.

## 2.3 Testing Error Conditions

It is generally not possible to safely change the execution order of statements in a group involving calls to `exit()` or `assert()`: hence compilers do not even try to optimize or parallelize the surrounding loops. So a good idea is to avoid these calls inside portions of code which are good candidates for optimization. However, it may not be desirable in all cases, in particular during the development phase of the sequential version of the application.

In particular, asserts are very useful for debugging the sequential version, but they can be easily removed by passing the `-DNDEBUG` option to the pre-processor, even if some compilers such as PIPS can take into account some of the information they carry as shown on Figure 2. So it is important to test the bounds of the entry parameters and the data not only to prevent some errors but also because it may give clues to prove that some pieces of code are indeed parallel. So instead of simply removing the asserts, they should be at least be replaced by some code that test some conditions of soundness.

Exits are often used when testing error conditions, such as system calls return values. But since these tests do not affect the program semantics when there is a correct execution, a good pratice would be to wrap system calls in dedicated macros or functions which would receive a simpler and non-blocking definition for the parallelizer, different from the real implementation that can be more complex, as discussed later in Section 5.2. Figure 3 gives an exemple for the `malloc()` function.

[2]Sometimes large constant values may lead to information loss because of overflow errors in the abstractions used to represent the programs internally. Symbolic bounds can be useful, even when they are

```
#include <assert.h>

int f(int m, int n) {
  int k = 0;
  assert(m >= 1 && n >= 1);
  /* From the previous assert,
     PIPS infers that m >= 1 and n >= 1 */
  for(int i = 0; i < m; i++)
    for(int j = 0; j < n; j++)
      k++;
  /* From the previous precondition after the assert,
     PIPS infers that k >= 1 at the function exit */
  return k;
}
```

Figure 2: Example of assert giving some semantics information in PIPS and PAR4ALL.

Another solution is to wrap error code between **#ifdef** `ERROR_CONTROL` ... **#endif** pairs to be able to remove such code on demand.

More generally, packaging this kind of complex I/O code into libraries that can also be parallelized by some specialists is a solution, as explained in Section 5.1.

## 2.4 Declaration Scope

Even in interprocedural optimizers which can deal with parameter passing through global variables such as COMMON in Fortran, or with static local variables, global declarations may induce memory dependencies which prevent parallelization. Removing these dependencies requires interprocedural program transformations which are not always possible nor available. They consist in cloning a whole program sub-tree and adding the global variable as parameters to all functions and function calls in the cloned sub-tree, resulting in more expensive analysis and transformation cost.

Hence, global (and Fortran `common`) or static local variables must be avoided at least in program parts which are good candidates for parallelization, and, preferably, in the whole application since program analyses and transformations more and more tend to take the whole program behavior into account.

C99 offers the possibility of declaring data almost anywhere in the code. Parallelizers are unequally ready to accept this feature, which means that you may have to gather all data declarations at the beginning of a statement block. This is nevertheless a good practice to avoid declarations intermingled with jumping constructs, as this may render code restructuring and maintenance more difficult.

---

actually constant.

```
// malloc wrapper with error testing
void * my_malloc(size_t size, int err_code,
                 const char * restrict err_mess)
{
  void * res = malloc(size);
  if (res == NULL) {
    /* This IO is an issue since it prevents
       parallelization: */
    fputs(err_mess, stderr);
    /* This global exit leads to an unstructured control
       graph, reducing optimization opportunities ... */
    exit(err_code);
  }
  return(res);
}

/* Wrapper provided to the parallelizer.
   Note that in case of a source-to-source parallelizer,
   yet another version may be given to the back-end
   compiler to take into account the targeted
   architecture characteristics. */
void * my_malloc(size_t size, int err_code,
                 const char * restrict err_mess)
{
  // No more IO or exit()!
  void * res = malloc(size);
  return(res);
}
```

Figure 3: Wrapping system calls.

## 2.5 Interprocedural Control Flow

Automatic code generation tools more and more include more or less advanced interprocedural capabilities, such as the propagation of constant values, summaries of scalar or arrays read and written by function calls (Cetus and PIPS, and even mixes of structures and arrays for PIPS). This greatly enhances their parallelizing capabilities.

A common restriction in parallelizing compilers and standard such as HMPP [8] or OpenACC [26] is that recursive functions are forbidden. The iterative version is then to be preferred, and, in some cases, this may even expose some intra-procedural instruction parallelism which could not be exploited with the recursive version because such optimization does not cross function calls.

# 3 Data Structures and Types

The C language provides several basic data structures, which can be combined to create more complicated ones. Traditionally, parallelizing compilers have focused their efforts on arrays, and this is still their preferred target. However, some are broadening their range of action to deal with mixes of structures and arrays, pointers... But it must be born in mind that aggregating data may reduce the optimization opportunities or their efficiency.

For instance, grouping the application infrastructure values inside a structure may be practical to avoid passing several parameters to functions, but it may prevent the compiler from using them when testing memory dependencies without data transformations.

Dealing with arrays of structures is similarly not practical, and dependency analysis can be easily extended to deal with them. However, generating communications for heterogeneous target is more difficult. Depending on the size of the structures, on the fields actually accessed during the exported computation, and on the communication costs, it may be more advisable to transfer a whole sub-part of the array, or to pack/unpack the desired fields to reduce the communication costs.

Notice also that unions are difficult to handle precisely. They create a discrepancy between memory usage and the actual semantics of the program since the same memory locations may be used to store completely unrelated objects. For safety reasons, the compiler must assume the dependencies, whereas the computations may be semantically independent. So unions, if needed, are to be kept in some purely sequential parts.

Finally, pointers are unequally handled. Analyzing and optimizing code using recursive data structures, such as linked lists or trees, is also still an active research area, for example with shape analysis. It may be useful to convert a linked list into an array if the processing cost is large enough. Another possibility for the compiler is to make use of speculative techniques to parallelize such codes.

The remainder of the section gives more specific advice to maximize the optimization opportunities for applications using pointers and arrays.

## 3.1 Pointers

Pointers allow to designate a single memory location using several names, or may designate several memory locations throughout its lifetime: this is called *dynamic aliasing*. However practical it may be, it makes subsequent program analyses more complicated, as compilers must safely assume that two pointers are aliased whenever they cannot prove the contrary. Some compilers provide ways to turn off this behavior when the user is sure that there is no pointer aliasing. For instance, Par4All provides a `--no-pointer-aliasing` command-line option. Other compilers such as the one from PGI requires the user to declare pointers with the C standard `restrict` qualifier. Basically, when using this qualifier, the programmer asserts that for the pointer lifetime, the pointed memory area is only accessed using this pointer. The compiler use this information when optimizing code since no memory access using another pointer will generate a dependence with the accesses using the restricted pointer. For example the OpenACC code on Figure 4 can be parallelized by the

```
void kernel(int n,
            float * restrict a,
            float * restrict r) {
#pragma acc region
  {
    for (int i = 0; i < n; ++i)
      r[i] = a[i]*2.0f;
  }
}
```

Figure 4: OpenACC example with the restrict qualifier required to allow parallelization.

PGI compiler only because the pointers are declared restricted.

Here are some pieces of advice to maximize the benefits of using source-to-source compilers allowing the use of pointers:

- Always use the same expression to refer to a memory location inside a function; for instance, avoid the following kind of code:

  ```
  my_parts = syst->domain[i]->parts;
  my_parts[j] = ...;
  ```

  since `syst->domain[i]->parts` is designated by 2 different expressions: `syst->domain[i]->parts` of course and `my_parts`, which may confuse the compiler tracking the memory use.

- Do not assign two different memory locations to a pointer (that means that a pointer should be considered rather as a single assignment variable). For example in the code

  ```
  if (cond())
    p = &a[f()];
  else
    p = &b[g()];
  *p = h();
  ```

  it is difficult for the compiler (and also a human being...) to figure out which element of which array is really written.

- Do not use pointer arithmetic, or solely with great care. In particular do not use it if it leads to pointer aliasing (avoid `p = p+i` or `p++` if `p` is a pointer). Note that this is a particular case of the previous point.

  For example avoid using pointers to perform a strength reduction on array accesses such as:

```
double a[N], b[N];
double *s = a, *d = b;
while(s<&a[N])
  *(d++) = *(s++);
```

but prefer the following clearer version that reflects the original algorithm and exposes some trivial parallelism:

```
double a[N], b[N];
for(int i = 0; i < N; i++)
  b[i] = a[i];
```

- Reserve the use of pointers for the sole dynamic allocation of arrays and to function parameter passing (in C, to emulate the C++ reference concept). In C++, use references instead of pointers, when possible.

  Note that you can have variable size arrays in C99 (as available in Fortran for decades), such as:

  ```
  int n = f();
  int m = g();
  double a[3*n][m+7];
  ```

  which can spare you the trouble of explicitly allocating and freeing pointers, or playing with `alloca()`.

- Avoid function pointers because, even when they are legal, they prevent precise interprocedural analyses and optimizations since it may be difficult to figure out which function is eventually called. This is also the case with virtual function in C++: the underlying implementation involves a table of function pointers.

- Do not use recursive data structures such as linked lists, trees... since it is often difficult to figure out what object is really pointed to, as explained previously in the introduction of the section.

- Be aware of the C standard constraints on pointer expressions, pointer differences, and more generally on pointer arithmetic.

## 3.2 Arrays

As array manipulations are often the source of massive parallelism, parallelizers often concentrate their efforts on them. However, their task can be made easier by following a number of coding guidelines which are detailed below.

### 3.2.1 Providing and Respecting Array Bounds

Parallelizing compilers often make the assumption that the program is correct, and in particular that array bounds are not violated. Otherwise, the behavior of the program may be undefined. This (bad) programming style is often used as array reshaping to iterate in a big global loop ranging all the elements normally visited by a loop nest on all dimension of an array, as shown in the next section. Modern compiler may figure out loop nest coalescing automatically to get maximum sequential performance anyway.

The bound should be provided

### 3.2.2 Linearization

In many automatic parallelizers, array references are represented using the integer polyhedron lattice; array reference indices must be affine integer expressions for the compiler to be able to study inter-iteration independencies, to propagate information over the program representation, and/or to generate communications.

For example `a[2*i-3+m][3*i-j+6*n]` is an affine array reference but `a[2*i*j][m*n-i+j]` is not (it is a polynomial of several variables).

This explains why you should not use array linearization to emulate accesses to multi-dimensional arrays with one-dimensional arrays, as in:

```
double a[n][m][l];
double * p = a;
for(int i = 0; i < n; i++)
  for(int j = 0; j < m; j++)
    for(int k = 0; k < l; k++)
      p[m*l*i + l*j + k] = ...;
```

The cleaner understandable following version should be used instead:

```
double a[n][m][l];
for(int i = 0; i < n; i++)
  for(int j = 0; j < m; j++)
    for(int k = 0; k < l; k++)
      a[i][j][k] = ...;
```

In the first cluttered version, the polynomial array index expression cannot be represented in a linear algebra framework; this kind of loop is usually not parallelized, and communications on GPU cannot be generated.

Some compilers (see [25, 13]) try to *delinearize* this kind of array accesses. But this transformation is not always successful.

### 3.2.3 Successive Array Element References

To reduce the analysis complexity, some compilers compute summaries of array element accesses in sets. For instance, PIPS array region analyses [11, 2] gather array elements in

```
/* In the following statement sequences and expression,
   all reference to a and b are made in a following way.
   So first the region with a[i-1] and a[i] is built,
    which is compact, and the a[i+1] reference is added,
   leading still to a compact region. */
tmp = a[i-1] + a[i] + a[i+1];
b[i-1] = ...;
b[i]   = ...;
b[i+1] = ...;
```

Figure 5: Consecutive array accesses.

```
/* The array region access is first computed from a[i-1]
    and a[i+1] which is non compact and then the a[i]
    element is added */
tmp = a[i-1] + a[i+1] + a[i];
/* The same for b */
b[i]   = ...;
b[i+1] = ...;
b[i-1] = ...;
```

Figure 6: Disjoint array accesses leading to imprecise array region analysis in PIPS and PAR4ALL.

sets represented by convex polyhedra. This means that non-convex sets of array elements are approximated by sets that contain elements which do not belong to the actual set, and are thus imprecise. Of course, further analyses and transformations are more likely to succeed and produce efficient code if array region analyses are more precise. So, in case of successive accesses to array elements, it is recommended to group them as much as possible so that two consecutive accesses in the program flow can be represented by a convex set. As an example, the version of Figure 5 is preferable to the version of Figure 6.

## 3.3 Casts

Casts are somewhat tricky to analyze because they induce translating a data memory layout into another one, and this maybe very difficult in the general case, especially for source-to-source tools which try to preserve the initial aspect of the program, and as the memory layout maybe architecture-dependent.

Hence the effects of the cast operator on the analyses may lead to a loss of precision: it is recommended to use it sparingly, out of the *hot* spots, and only when it does not impact the memory layout.

# 4 Directives and Pragma (OpenMP...)

Fashionable extensions to sequential languages are hints expressed as pragma or directives hidden into comments in the source code to inform the compiler that some parts can be executed in parallel, offloaded to an accelerator or with data and computation distributed on a distributed-memory massively parallel computer. The extensions we are interested in here are those which do not modify the sequential semantics of a program. For example, expressing that a sequential part is parallel does not change the semantics of the sequential part and this extension can be ignored by a compiler and the program remains correct[3].

OpenMP [27, 28] and HPF [17, 18] are old standards based on this principle, but with different targets and paradigms. OpenMP targets shared-memory multiprocessors whereas HPF is more ambitiously oriented towards distributed-memory massively parallel machines with deeper compiler support. Unfortunately, only OpenMP has been successful in its achievements, even if HPF had some impact on the development of parallel models and languages [21]. Figure 7 shows an example of OpenMP.

More modern extensions such as HMPP [8] or OpenACC [26] are developed for C, C++ and Fortran to extend these concepts to other domains, such as heterogeneous computing or even to embedded systems with SMEC [3]. An example of OpenACC Fortran program is found in Figure 8.

The interesting aspect is that even if the compiler dealing with this extension is not or no longer available, the sequential program remains and is not lost. It can still be optimized and parallelized later, manually or automatically with other tools.

It is an interesting path toward incremental manual parallelization without loosing the investments in the source code that remains sequential. But the sequential sources need to be reorganized to express parallelism exploitable by the tools. Thus this extensions take advantage of the good practices described in this chapter.

# 5 Using Libraries

## 5.1 Relying on Efficient Libraries

An easy way to parallelize part of the execution in a sequential program is to use some libraries that are already parallelized, such as linear algebra libraries (MKL, BLAS, LAPACK, PETSc...) or other mathematical libraries (FFTW [14] to compute various fast Fourier's transforms...).

Even I/O operations exists as parallel libraries, which is of great importance since computer performance increase quicker that I/O latency reduction and bandwidth improvement.

In this way, a programmer can benefit from highly parallelized and optimized expertise packaged in libraries (ScaLAPACK, PLASMA, MAGMA, CuBLAS, PETSc...) without having

[3]Actually the fine operational semantics may change because for example of the non-associativity of floating-point computations or some signed integer operations due to various rounding.

```
/* This is executed by several threads concurrently */
#pragma omp parallel for
  for (i = 0; i < n; i++)
      // Iterations are distributed between the threads
      x[i] += y[i];
  // Implicit synchronization here

// Launch all the threads
#pragma omp parallel
{
  /* But the following is run on only one thread,
     with its own copy of p */
  #pragma omp single private(p)
  {
    // Iterate on the elements of a list from the head
    p = listhead;
    while (p) {
      /* While there is still an element, launch a new
         task asynchronously to process it in parallel */
      #pragma omp task
      {
        process (p);
      }
      // Look at next element
      p = next(p);
    }
  }
}
```

Figure 7: Small example of C language with OpenMP extension.

to invest time in the parallelization phase.

Unfortunately, not all the parts of an application exists as libraries, and if so, may not be combined. Furthermore, they may not be directly usable with some automatic parallelizer, as explained in the following section.

## 5.2 Quarantined Functions: Stubs

Using complex functions can stress too much some compilers or sometimes, parallelizing a program globally need the whole program source but the source of the libraries is not available or definitely too complex or irrelevant to be retargetable.

In this case, two different compilers may be needed: one to do the automatic par-

```
  ! Explain that Anew is to be allocated to the accelerator.
  ! A has a copy also allocated to the accelerator and
  ! is initialized with the value of A from the host:
!$acc data copyin(A), create(Anew)
iter = 0
do while ( err .gt. tol .and. iter .gt. iter_max )
  iter = iter + 1
  err = 0.0
  ! The following loop nest is parallel and to be outlined
  ! for execution on the accelerator with some scheduling
  ! parameters.
  ! Instruct also the compiler there is a reduction done
  ! on the err variable
!$acc kernels loop reduction(max:err), gang(32), worker(8)
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * ( A(i+1,j ) + A(i-1,j ) &
                           A(i, j-1) + A(i, j+1) )
      err = max( err, abs(Anew(i,j) - A(i,j)) )
    end do
  end do
!$acc end kernels loop
  if( mod(iter, 100) .eq. 0 ) print*, iter, err
  ! This affectation can be done in parallel,
  ! as with HPF workload:
!$acc parallel
  A = Anew
!$acc end parallel
end do
!$acc end data
```

Figure 8: Example of OpenACC with Fortran 90 commented after [16].

allelization on a code with some functions hidden from it and another more robust non parallelizing compiler to compile the problematic functions. Afterwards, the two programs are linked together to produce a global program.

But there may be still an issue: if the parallelization is performed by program comprehension, for example by using some abstract interpretation like in PIPS, the whole program source has to be analyzed, with all the library functions which are unfortunately unavailable. A description of the behaviors of these functions is needed for automatic parallelization.

This can be provided by some external ways such as a contract description in an XML

files such as with SPEAR tools [23], by using some **#pragma** in HMPP [8] or OpenACC [26], directives or attributes such as in Fortran (`intent(in)`, `intent(out)` or `intent(inout)` specifying some arguments are read or/and written by a function).

Another way is to provide the parallelizer with a stub function that mimic the behavior needed by the parallelizer to know what is needed for a parallelism analysis. For example in PAR4ALL, which is based on PIPS, the memory accesses to the memory are analyzed to know if there are conflicts or not between pieces of code, scalar variable values are tracked by semantic analysis for improving dependency test and applying various optimizations, input/output effects are tracked to avoid unfortunate parallelism of input/output and so on. So for PAR4ALL, stub functions with these effects are needed.

The advantage of this last solution is that no language extension or **#pragma** is needed. The drawback is that some simple functions have to be written instead and the global compilation flow has to be adapted to provide the stub functions to the parallelizer and to substitute afterwards in the final executable the real functions.

# 6 Object Oriented Programming

Although we focus in this article on the C and Fortran languages, object-oriented programming gain momentum even in high-performance computing, with C++ which is an object-oriented extension of the C language, or even more logically Fortran 2003 and 2008 versions that include directly object-oriented concepts.

Object-oriented languages have some interesting high-level aspects allowing conciseness and abstraction with the drawback that it may difficult to understand what is really executed on the target, by comparison with simpler languages as C, closer to a high-level macro-assembler.

As for languages that are not object-oriented , a simple general rule is to avoid using constructs that leads to unstructured code (such as exceptions and object creation/deletion) and unpredictable code (such as virtual functions) in compute intensive parts with some parallel potential such as heavy loop nests.

Using virtual functions in C++ means that the choice of the real function to be used is only made at execution time, which is costly (equivalent to using a function pointer in C) and difficult to analyze by a parallelizer compiler.

When possible, using advanced templates (which can be seen as a kind of high-level C preprocessor) replaces virtual functions by compile-time specialization with simpler function overloading which are plain function calls. They can even be in-lined.

Furthermore, using C++ well-known templates, for example from the STL or BOOST libraries, may be recognized by the compiler. For example, by understanding its semantics, a loop iterating on a `std::vector` is replaced by some parallel constructs [24]. There are even some direct implementations of the algorithms from STL that are directly parallel, such as the MCSTL [35] or STAPL [7], or variations with some specialized parallel template libraries such as TBB [19].

Pointers used in C to pass arguments in functions can be replaced by reference (`&`)

in C++ to precise the programmer intentions and to let the parallelizer be able to find parallelism.

Working with expressions on objects in C++ is quite powerful but may create a lot of temporary objects with a lot of time spent in object creation and deletion, content copy. This may spoil memory bandwidth and electrical power, but also have some side effect preventing the parallelization. A nice feature from C++11 is that there are some move constructors that may avoid some useless object copy or creation with their side effects.

Exceptions (or arbitrary jump) are another side effect to avoid in the parallel sections of a program. Even if they are caught locally, it is equivalent to unstructured control flow that may impair the parallelism. More generalized use of exceptions that may be caught interprocedurally for example can be seen as non-local **goto**[4] which is inherently challenging but question the parallelism itself: what is the semantics of a parallel loop with an exception in an iteration? Do we rollback all the loop in a transaction way? Do we execute all the iterations but not the trouble-making one? Do we execute all the iteration up to the exception-throwing one to emulate a sequential semantics?

# 7 Conclusion

Agent Smith:

> *Never send a human to do a machine's job.*
>
> In *Matrix* (Andy & Larry Wachowski, 1999).

Because of the current and foreseeing technological constraints, parallelism is the only way to go for speed and energy efficiency. Unfortunately, parallel programming has been a real challenge for more than 50 years now.

Parallelizing compilers are promising tools to generate code for a variety of architectures. Of course, the generated code is often not as efficient as hand-programming by specialists, neither as clean as a new development in a pure parallel programming language, but is an interesting trade-off relative to the time-to-market advantage for legacy code.

Automatic parallelization is no magic and has been a research area for 40 years. Unfortunately it has not became mainstream, perhaps because of too high expectations. Parallelization is an intractable issue in general and one cannot expect an automatic tool to solve better intractable problems. Fortunately, providing some rules are obeyed when writing an application, parallelism becomes easier to detect by parallelizing compilers. Low hanging fruits must be picked first.

Even if the rules presented here have been devised based upon experience with pips and Par4All, they are far more general and even applicable to modern compilers targeting sequential targets that have more and more parallelism anyway, such as simd instruction sets.

[4] Like `setjmp()` and `longjmp()` in C.

Basically, the application code should be as structured as possible. At least the unstructured parts should be segregated into some parts not to be parallelized, to keep the time-consuming parallelized parts large enough.

Easy parallelized parts often use regular **for** loops like Fortran **do** loops, preferably to **while** loops. Loop nest candidates for parallelization should not contain I/O or debug or error control code, or there would be an easy way to switch them off.

Automatic parallelization implies some kind of program comprehension and everything that uselessly obfuscates the program design is to be banned. Using clear data types, without type casts, local declaration instead of long-life data structures, explicit argument passing instead of global variable side effects are some of the basic rules to follow.

Since pointers are often not well managed by a compiler because of the difficulty to track what is really pointed to, pointer use has to be minimized as the use of recursive data structures, function pointers or virtual function in object-oriented languages. As a particular case, avoid array linearization and casting, but prefer using the data as they are declared. Modern compilers do no longer need these kind of manual optimization to get good sequential performance, so good sequential programming does not impact execution times usually.

If some of these recommendations cannot be applied, then try to group the non-compliant code outside of loop nests which are good candidates for parallelization or, more generally speaking outside of the most computational-intensive parts of the program. This is a good practice, for instance, for heap allocations.

In some case, the programmer has some knowledge on the program that is difficult to figure out automatically. Attributes in some languages, decoration or pragmas can be used to help the compiler to parallelize the code. When these pragma do not change the sequential semantics as with OpenMP, OpenHMPP or OpenACC, this is an efficient pragmatic attitude to go further into a parallel execution without losing the sequential program, and thus without having to maintain different program versions alive.

But before launching heavy parallel developments, relying on some libraries already parallelized by some experts for a large range of platform is definitely the way to begin with. Unfortunately the original application may not be structured to use an existing library but this restructuring should be worth the investment.

Actually, most of the previous rules are some common sense rules that are more and more common with the development of recent new languages or even with recent versions of existing languages, such as C++11 or Fortran 2008. So good programming for parallelism begins with good sequential programming.

Even if these programming rules seem constraining, they are often considered as sound programming rules even for classical sequential programming. That means that (re)writing application to ease parallelization can be a good opportunity for code cleaning and modernization by reengineering.

From a higher point of view, program parallelization implies some language and tool choices. Since the program sources are the real value of the applications, one should use standard approaches, if possible Open Source, to be more confident in the life duration of the environment and to not being hijacked by some companies and be bound to their future.

When parallelizing a program, the entry cost is often important, because the program has to be fitted into some new constructs, may be rewritten in a new language with a new high-level architecture. But if some of the tool used disappear for some reason, the exit cost may be quite worse, because reverse engineering the optimizations that have been made to fit the former environment before even thinking to port the all application to a new environment can be a nightmare.

For these reasons, a milder approach involving some coding rules and automatic parallelization, with some optional pragmas or directives and some use of parallel libraries, seems a good compromise between efficiency, time-to-market and technology continuity.

# 8 Acknowledgments

This work is supported by the OpenGPU project with funding from the French Systém@TIC Research Cluster.

We would like to thanks all the PIPS and Par4All team, mainly at MINES ParisTech and SILKAN, and all the members of the OpenGPU project for their fruitful discussions and comments.

# References

[1] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, P. Villalon, "PIPS Is not (just) Polyhedral Software", in *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*. Chamonix, France, Apr. 2011, `http://perso.ens-lyon.fr/christophe.alias/impact2011/impact-09.pdf`.

[2] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J.O. McMahon, F.X. Pasquier, G. Péan, P. Villalon, "Par4All: From Convex Array Regions to Heterogeneous Computing", in *2nd International Workshop on Polyhedral Compilation Techniques (IMPACT 2012)*. Paris, France, Jan. 2012.

[3] R. Barrère, M. Beemster, R. Keryell, "SME-C — C99 with pragma and API for parallel execution, streaming, processor mapping and communication generation", Technical report, SMECY Project, 2012.

[4] M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs", in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS, pages 225–234. ACM, New York, NY, USA, 2008, ISBN 978-1-60558-158-3.

[5] C. Bastoul, *Improving Data Locality in Static Control Programs*, PhD thesis, Université Paris VI, Dec. 2004.

[6] M.W. Benabderrahmane, L.N. Pouchet, A. Cohen, C. Bastoul, "The polyhedral model is more widely applicable than you think", in *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303. Springer-Verlag, Berlin, Heidelberg, 2010, ISBN 3-642-11969-7, 978-3-642-11969-9, URL `http://dx.doi.org/10.1007/978-3-642-11970-5_16`.

[7] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N.M. Amato, L. Rauchwerger, "STAPL: standard template adaptive parallel library", in *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10. ACM, New York, NY, USA, 2010, ISBN 978-1-60558-908-4, URL `http://doi.acm.org/10.1145/1815695.1815713`.

[8] CAPS Entreprise, "HMPP Workbench", `http://www.caps-entreprise.com`, 2010.

[9] C. Chen, J. Chame, M. Hall, "CHiLL: A framework for composing high-level loop transformations", Technical Report 08-897, University of Southern California, June 2008.

[10] B. Creusillet, *Array Region Analyses and Applications*, PhD thesis, École des Mines de Paris, Dec. 1996, Available at `http://www.cri.ensmp.fr/doc/A-295.ps.gz`.

[11] B. Creusillet, F. Irigoin, "Interprocedural analyses of Fortran programs", *Parallel Computing*, 24(3–4): 629–648, 1998, URL `http://citeseer.ist.psu.edu/creusillet97interprocedural.html`.

[12] P. Feautrier, "Dataflow Analysis of Array and Scalar References", *International Journal of Parallel Programming*, 20(1): 23–53, Sept. 1991.

[13] B. Franke, M. O'Boyle, "Array recovery and high-level transformations for DSP applications", *ACM Trans. Embed. Comput. Syst.*, 2(2): 132–162, May 2003, ISSN 1539-9087, URL `http://doi.acm.org/10.1145/643470.643472`.

[14] M. Frigo, S.G. Johnson, "The Design and Implementation of FFTW3", *Proceedings of the IEEE*, 93(2): 216–231, 2005, Special issue on "Program Generation, Optimization, and Platform Adaptation".

[15] M. Griebl, *The mechanical parallelization of loop nests containing while loops*, PhD thesis, Passau, Passau, 1997.

[16] M. Harris, "An OpenACC Example (Part 2)", `http://developer.nvidia.com/cuda/openacc-example-part-2`, 2012.

[17] High Performance Fortran Forum, "High Performance Fortran Language Specification", Version 1.0 CRPC-TR 92225, Center for Research on Parallel Computation, Rice University, Houston, USA, May 1993, `http://hpff.rice.edu/versions/hpf1`.

[18] High Performance Fortran Forum, "High Performance Fortran Language Specification", Version 2.0, Center for Research on Parallel Computation, Rice University, Houston, USA, Jan. 1997, http://hpff.rice.edu/versions/hpf2.

[19] Intel, "Intel Threading Building Blocks for Open Source (Intel TBB)", http://threadingbuildingblocks.org, 2012.

[20] F. Irigoin, P. Jouvelot, R. Triolet, "Semantical Interprocedural Parallelization: An Overview of the PIPS project", in *International Conference on Supercomputing*, pages 144–151, June 1991.

[21] K. Kennedy, C. Koelbel, H. Zima, "The rise and fall of high performance Fortran", *Communications of the ACM*, 54(11): 74–82, Nov. 2011, ISSN 0001-0782, URL http://doi.acm.org/10.1145/2018396.2018415.

[22] C. Lengauer, "Loop Parallelization in the Polytope Model", in *Proceedings of the 4th International Conference on Concurrency Theory*, CONCUR '93, pages 398–416. Springer-Verlag, London, UK, UK, 1993.

[23] E. Lenormand, G. Édelin, "An industrial perspective: A pragmatic high-end signal processing design environment at THALES", in *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*, 2003.

[24] C. Liao, D.J. Quinlan, J.J. Willcock, T. Panas, "Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore", in *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 28–41. Springer-Verlag, Berlin, Heidelberg, 2009, ISBN 978-3-642-02284-5, URL http://dx.doi.org/10.1007/978-3-642-02303-3_3.

[25] V. Maslov, "Delinearization: an efficient way to break multiloop dependence equations", *SIGPLAN Not.*, 27(7): 152–161, July 1992, ISSN 0362-1340, URL http://doi.acm.org/10.1145/143103.143130.

[26] O. Members, "The OpenACC™ Application Programming Interface", Version 1.0, OpenACC Members, Nov. 2011, http://www.openacc-standard.org.

[27] OpenMP Architecture Review Board, *OpenMP Standard, 1.0*, Oct. 1997, http://www.openmp.org.

[28] OpenMP Architecture Review Board, *OpenMP Standard, 3.1*, July 2011, http://www.openmp.org.

[29] The Portland Group, *PGI Compiler User's Guide*, 2012.

[30] L.N. Pouchet, *PoCC: the Polyhedral Compiler Collection Package*, 2012, http://www.cse.ohio-state.edu/~pouchet/software/pocc/doc/pocc.pdf.

[31] Purdue University, *The CETUS Compiler Manual*, 2011, URL `http://cetus.ecn.purdue.edu/Documentation/manual`.

[32] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, Q. Yi, *ROSE User Manual: A Tool for Building Source-to-Source Translators, Draft User Manual*, Lawrence Livermore National Laboratory, June 2012.

[33] Reservoir Labs, Inc., *R-Stream Parallelizing C Compiler, Power User Guide*, 2012.

[34] SILKAN, "Par4All — Single Source, Multiple Targets", 2012, `http://par4all.org`.

[35] J. Singler, P. Sanders, F. Putze, "MCSTL: The Multi-core Standard Template Library", in A.M. Kermarrec, L. Bougé, T. Priol (Editors), *Euro-Par 2007 Parallel Processing*, Volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer Berlin / Heidelberg, 2007, ISBN 978-3-540-74465-8, URL `http://dx.doi.org/10.1007/978-3-540-74466-5_72`.

[36] E. Stoltz, "Effective Compiler Utilization", in *Linux Supercluster User's Conference*, Sept. 2000.

[37] R. Triolet, P. Feautrier, F. Irigoin, "Direct Parallelization of Call Statements", in *ACM SIGPLAN Symposium on Compiler Construction*, pages 176–185, 1986.