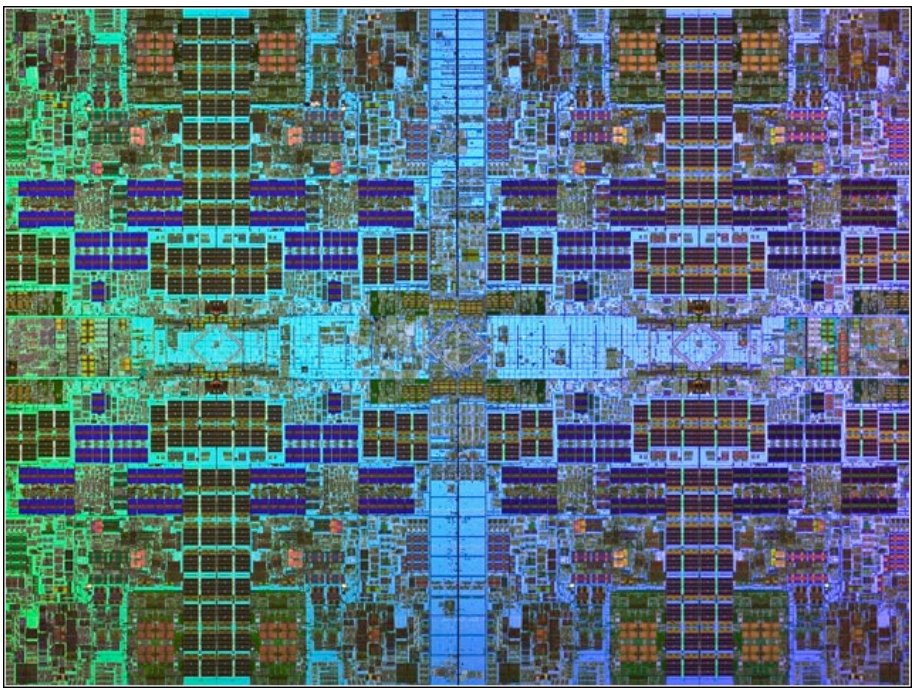
A parallelization framework
for numerical simulation

# The LPlib library



Loïc MARÉCHAL / INRIA, Gamma Project

January 2016
Document v1.7

# Contents

Cover pictures : X-ray view of an IBM Power7 octo-core processor.

# 1 Introduction

The purpose of the *LPlib* library (each *italicized* expression is defined in the glossary at the end of this document) is to provide programmers of solvers or automated meshing software in the field of scientific computing with an easy, fast and transparent way to parallelize their codes.

This library is based on posix standard threads (posix-threads [1] also known as *pthreads*) thus taking advantage of *multi-core* chips and *shared memory* architectures supported by most platforms (Linux, macOS, Unix, Windows).

This document is just about integrating the library into existing programs. Only a slight overview of underlying technologies will be given.

## 1.1 Motivation

The initial motivation was to take advantage of today's ubiquitous multi-core computers.

Since 2004, first Moore's law corollary has plummeted from the 40% yearly increase in processor frequency, it enjoyed for the last 30 years, to a meager 10%.

As for now, speed improvement can only be achieved through the multiplications of processors (now called cores) within a single chip.

This adds a new burden to programmers : while speed increase through higher frequency implied little code modifications (notably cache-miss minimization), multi-core chips require the code to be parallelized, and sometimes, thoroughly rewritten !

## 1.2 A.P.I.

Direct call to pthreads commands is still too tedious and non-transparent from a user's point of view, so, an encapsulation providing a simplified *A.P.I.* could be helpful.

The *LPlib* allows for easy parallelizing of loops running over tables and structures featuring direct or indirect memory accesses. These account for most of the computing time in numerical simulation software dealing with meshes.

A programmer does not need to rewrite his code but only needs to encapsulate loops into independent procedures and briefly describe the data structure they will run over (such as the size of the tables and whether they will be accessed directly or not).

Right now, this library can only be integrated into any C program, but a Fortran API is under development.

## 1.3 Features

The *LPlib* main feature is its ability to deal with indirect memory accesses. This kind of memory access is widely used when working on meshes and prevents automatic parallelization capabilities of modern compilers.

The problem is handled through a mesh renumbering scheme using a Peano-Hilbert curve [3] or any renumbering scheme which increases geometric compactness, thus minimizing memory gaps between neighboring entities in a mesh. A mesh renumbered through such a curve would feature a much greater intrinsic degree of parallelism.

This library also allows for asynchronous processes to be launched, enabling procedures pipelining, a very useful technic when it comes to input/output or data pre-processing steps independent from each other.

# 2 Usage

## 2.1 Installation and compilation

The *LPlib* may be used from any software written in ANSI C language.

It is made of two files : lplib3.c that has to be compiled and linked along with the calling code, and a header file, lplib3.h, to be included by any source file using *LPlib* commands.

A software environment providing pthreads is needed (most Unix-like are compatible). Practically speaking, you only need to locate the files pthreads.h and libpthreads.so and to provide their paths to the compiler and linker.

Use "-I /path_to_headers" to tell the compiler where to locate the pthreads.h file and pass "-L /path_to_libraries" and -lpthreads options to the linker.

## 2.2 Initialization

The *LPlib* must be initialized only once before calling any library function. It is easily done by calling the InitParallel command and providing it with the maximum number of processors to be used (this number may be lower or greater than the actual number of physical processors for testing purposes)

```
IndexOfInstance = InitParallel(NumberOfProcessors);
```

Conversely, the *LPlib* should be properly closed at the end of the program :

```
StopParallel(IndexOfInstance);
```

in order to free the library memory and kill all threads.

Since real life examples are more telling than any academic discussion, the way to use *LPlib* will be introduced via three examples of increasing complexity.

## 2.3 Example 1 : computing the mean value for each triangle of a mesh

### 2.3.1 Statement

Let's say we have a *mesh* stored in a structure of type $MeshStruct$, comprising a table $VerTab$ of $nbv$ vertices and a table $TriTab$ of $nbt$ triangles.

```
typedef struct
{
    double coordinates[2];
    double t; /* temperature */
    int c;    /* counter */
    int num;  /* index of vertex in VerTab */
}VerStruct;

typedef struct
{
    VerStruct *VerTab[3];
    double t; /* temperature */
}TriStruct;

typedef struct
{
    /* global maximum value and a table of local maximums */
    double maximum, maximums[ NumberOfProcessors ];
    int nbv, nbt;
    VerStruct VerTab[ nbv ];
    TriStruct TriTab[ nbt ];
}MeshStruct;
```

Temperature is stored at each vertex. We wish to compute a triangle temperature as the mean value of its three vertices. The program is going to loop over triangles, reading their vertex temperatures (indirect read access) and writing the resulting value into the triangle structures themselves (direct write access).

Since data is written in the same structure on which the code is looping over, the loop is a direct memory access one.

### 2.3.2 Serial program

```
main()
{
    for(i=0;i<mesh->nbt;i++)
    {
        mesh->TriTab[i]->t = 0;

        for(j=0;j<3;j++)
            mesh->TriTab[i]->t += mesh->TriTab[i]->VerTab[j]->t;

        mesh->TriTab[i]->t /= 3;
    }
}
```

### 2.3.3 Parallel program

```
main()
{
    /* Initialize LPlib with 4 processors */
    LibIdx = InitParallel(4);

    /* Declare a new table of nbt triangles */
    TriType = NewType(LibIdx, mesh->nbt);

    /* Launch the parallelized loop computing the average values */
    LaunchParallel(LibIdx, TriType, 0, ComputeAverage, (void *)mesh);

    /* Close LPlib */
    StopParallel(LibIdx);
}

void ComputeAverage(int begin, int end, int thread, void *arguments)
{
    /* Fetch arguments from the single structure */

    MeshStruct *mesh = (MeshStruct *)arguments;

    /* The begining and ending loop indices are imposed by the scheduler.
       The rest of the loop remains unaffected. */

    for(i=begin; i<end; i++)
    {
        mesh->TriTab[i]->t = 0;

        for(j=0;j<3;j++)
            mesh->TriTab[i]->t += mesh->TriTab[i]->VerTab[j]->t;

        mesh->TriTab[i]->t /= 3;
    }
}
```

The LaunchParallel command will launch concurrently four occurrences of the ComputeAverage procedure providing each one with a different set of beginning and ending indices (let's say $nbt = 1000$ in this example) :

```
ComputeAverage(  0, 249, 0, mesh);
ComputeAverage(250, 499, 1, mesh);
ComputeAverage(500, 749, 2, mesh);
```

```
ComputeAverage(750, 999, 3, mesh);
```

## 2.4 Example 2 : searching for the maximum value among a set of triangles

### 2.4.1 Statement

Now, we would like to find out which triangle has the greatest temperature value.

### 2.4.2 Serial program

```
main()
{
    mesh->maximum = -273.15;

    for(i=0;i<mesh->nbt;i++)
        if(mesh->TriTab[i]->t > mesh->maximum)
            mesh->maximum = mesh->TriTab[i]->t;
}
```

### 2.4.3 Issues

The parallel version needs special care. If all threads were to write at the same time to a global variable called *maximum*, the memory access conflict would result in a wrong final maximum temperature.

A workaround is to set up a little maximum temperatures table (one entry per thread) where each thread would store its own local maximum value. A final serial loop would eventually compute the global maximum value from local ones (even though this loop is serial, it is only run over the number of threads).

### 2.4.4 Parallel program

```
main()
{
    /* Launch the parallel part of the code */

    LibIdx = InitParallel(4);
    TriType = NewType(LibIdx, mesh->nbt);
    LaunchParallel(LibIdx, TriType, 0, ComputeMaximum, (void *)mesh);
    StopParallel(LibIdx);

    /* Perform a loop on the table of local average values computed
       by each thread in order to find the global maximum value. */
```

```
    mesh->maximum = -273.15;

    for(i=0;i<4;i++)
        if(mesh->maximums[i] > mesh->maximum)
            mesh->maximum = mesh->maximums[i];
}

void ComputeMaximum(int begin, int end, int thread, void *arguments)
{
    MeshStruct *mesh = (MeshStruct *)arguments;

    /* The local maximum value is stored in the maximums' table
       under the index given by the thread number */

    mesh->maximums[ thread ] = -273.15;

    for(i=begin; i<end; i++)
        if(mesh->TriTab[i]->t > mesh->maximums[ thread ])
            mesh->maximums[ thread ] = mesh->TriTab[i]->t;
}
```

The above code will perfectly work but may get slower and slower as the number of threads increases. So the following modified version should be preferred:

```
void ComputeMaximum(int begin, int end, int thread, void *arguments)
{
    MeshStruct *mesh = (MeshStruct *)arguments;
    double thread_maximum = mesh->maximums[ thread ];

    /* The local maximum value is stored in the maximums' table
       under the index given by the thread number */

    mesh->maximums[ thread ] = -273.15;

    for(i=begin; i<end; i++)
        if(mesh->TriTab[i]->t > thread_maximum)
            thread_maximum = mesh->TriTab[i]->t;

    mesh->maximums[ thread ] = thread_maximum;
}
```

Why copy each thread maximum value from the globally accessible table maximums[] to a variable local to the thread ? You could do without these copies since each thread

would only access its own bucket and no memory write contention should occur. But doing so would generate a lot of memory access called "false sharing" that would not jeopardize the result validity but greatly reduce the code efficiency.

Indeed, all table values are stored consecutively in memory and as such, would very likely reside within the same cache line. Each core calculating the maximum value would then have to load this cache line in its own level one cache and each time it would write a new maximum value, it would have to tell all other cores to reload the full line. This cache reload is indeed useless since a thread would never modify the maximum values of others, but it has no way to know about it. Consequently, each write access to the shared cache line will generate as many cache reload as there are cores, thus the total number of memory access will grow with the square of the number of threads !

Dealing with such problem is very straightforward: when computing a thread local vector reduction (sum, maximum or residual value), the right way is to copy the thread's value from the global table to a local variable then perform the reduction loop on the local variable and finally copy the resulting value back to the global table, thus avoiding the false sharing.

## 2.5 Example 3 : an indirect memory access loop featuring a cross dependency problem

### 2.5.1 Statement

Using meshes and matrices in scientific software leads inevitably to more complex algorithms where indirect memory accesses are needed.

For example, accessing a vertex structure through a triangle link in the following instruction :

```
TriTab[i]->VerTab[j];
```

Such memory access is very common in C, the compiler will first look for a triangle $i$, then a vertex $j$, thus accessing the data indirectly.

If a loop is done over triangles, adding their temperature to their three vertices' own temperature value, it would lead to a writing conflict. Such algorithm is made of three steps:

1. clear each vertex temperature
2. loop over triangles : add each triangle temperature to its three vertices temperature and increase vertices' counter.
3. loop over vertices : divide the accumulated temperature value by the number of contributing triangles.

### 2.5.2 Serial program

```
main()
{
    /* Clear the temperature and counter associated to each vertex */

    for(i=0;i<nbv;i++)
    {
        mesh->VerTab[i]->t = 0;
        mesh->VerTab[i]->c = 0;
    }

    /* main loop adding the triangle temperature to each of
       its vertex temperature and incrementing each vertex counter
       in order to keep track of the number of triangles contributing
       to each vertices temperature */

    for(i=0;i<mesh->nbt;i++)
        for(j=0;j<3;j++)
        {
            mesh->TriTab[i]->VerTab[j]->t += mesh->TriTab[i]->t;
            mesh->TriTab[i]->VerTab[j]->c++;
        }

    /* divide each vertex temperature by the number
       of contributing triangles */

    for(i=0;i<nbv;i++)
        mesh->VerTab[i]->t /= mesh->VerTab[i]->c;

}
```

### 2.5.3 Issues

If no attention is paid to memory conflicts, the *LPlib* partitioner will split the triangle table into four same sized blocks and assign them to four threads :

```
ComputeAverage(  0, 249, 0, mesh);
ComputeAverage(250, 499, 1, mesh);
ComputeAverage(500, 749, 2, mesh);
ComputeAverage(750, 999, 3, mesh);
```

Even though, there are no two blocks sharing the same triangles, it is not the case when it comes to vertices. For example, triangle 250 from block 1 and triangle 750 form block 3

may share the same vertex. If the two threads where to process those triangles at the same time, they would write different temperatures at the same memory location, resulting in an undetermined value.

To handle such a situation properly, the *LPlib* must be provided with the mesh connectivity. That is the list of vertices pointed to by triangles. This way, the library will split the triangles table into a greater number of chunks (typically ten blocks per threads), and compute the dependency between those blocks.

The *LPlib* will provide each thread with blocks that not only don't share common triangles, but whose triangles don't share common vertices. Only this way all threads could run together.

### 2.5.4 Parallel program

```
main()
{
    LibIdx = InitParallel(4);

    /* Define types and their dependencies */

    TriType = NewType(LibIdx, mesh->nbt);
    VerType = NewType(LibIdx, mesh->nbv);

    BeginDependency(LibIdx, TriType, VerType);

    for(i=0;i<mesh->nbt;i++)
        for(j=0;j<3;j++)
            AddDependency(LibIdx, i, mesh->TriTab[i]->VerTab[j]->num);

    EndDependency(LibIdx);

    /* Launch all three parallel loops */

    LaunchParallel(LibIdx, VerType,       0, ClearTemperature, (void *)mesh);
    LaunchParallel(LibIdx, TriType, VerType, AddTemperature,   (void *)mesh);
    LaunchParallel(LibIdx, VerType,       0, ScaleTemperature, (void *)mesh);

    StopParallel(LibIdx);
}

/* Loop clearing temperatures and counter in parallel */

void ClearTemperature(int begin, int end, int thread, void *arguments)
{
```

```
    MeshStruct *mesh = (MeshStruct *)arguments;

    for(i=begin; i<end; i++)
    {
        mesh->TriTab[i]->t = 0;
        mesh->TriTab[i]->c = 0;
    }
}

/* Loop adding a triangle temperature to its vertices in parallel */

void AddTemperature(int begin, int end, int thread, void *arguments)
{
    MeshStruct *mesh = (MeshStruct *)arguments;

    for(i=begin; i<end; i++)
        for(j=0;j<3;j++)
        {
            mesh->TriTab[i]->VerTab[j]->t += mesh->TriTab[i]->t;
            mesh->TriTab[i]->VerTab[j]->c++;
        }
}

/* Loop dividing vertices temperature by the number
   of contributing triangles */

void ScaleTemperature(int begin, int end, int thread, void *arguments)
{
    MeshStruct *mesh = (MeshStruct *)arguments;

    for(i=begin; i<end; i++)
        mesh->VerTab[i]->t /= mesh->VerTab[i]->c;
}
```

## 2.6 Dynamic data modifications

Dealing with dynamic data, either topology changes or table growing, is challenging for parallel computing. This problem is partially addressed by the *LPlib* with the following commands: UpdateDependency and ResizeType.

There is no means to remove dependencies on the fly, you can only add up more of them with the command *UpdateDependency*, which can be called only outside a parallel loop. It could be useful when creating new elements with new dependencies albeit the old ones will remain, thus impeding the parallelism efficiency. When doing heavy topological

modifications to a mesh, it is advised to check periodically the dependency statistics with the *GetDependencyStats* command and to completely free and reallocate the whole mesh data types and dependencies when the average number of collisions grows beyond 1/5 the number of threads.

Likewise, a data type may be resized outside a parallel loop, but only to make it grow, not to shrink it. Note that due to initial static over-allocation, the resized table may not increase more than two folds against its initial size. In such a case, you would have to free and rebuild the whole data types and dependencies.

Both methods allow only for lightweight modifications to be applied to the mesh without having to rebuild the whole dependencies between each parallel loop, which can be very costly and may spoil the whole parallelization gain.

## 2.7 Software pipelining

So far, we've only used symmetric parallelism, a scheme where every thread is executing the same piece of code but on different sets of data (typically each thread applies the same function to a vector subset).

Asymmetrical parallelism is a complementary technic where processors execute totally different codes working on different and independent data.

Since the main body of most programs runs like this:

```
main()
{
    procedure1(data1);
    procedure2(data2);
    procedure3(data3);
    ...
}
```

As long as a procedure does not need any data resulting from a previous one, or no data it needs has been modified, it may be launched concurrently without waiting for the completion of former procedures.

In the above example, if procedures 1 and 2 are completely independent from each other, they may be launched concurrently via the LaunchPipeline command:

```
main()
{
    ProcIndex1 = LaunchPipeline(LibIndex, procedure1, data1, 0, NULL);
    ProcIndex2 = LaunchPipeline(LibIndex, procedure2, data2, 0, NULL);
    ...
}
```

Unfortunately, many procedures depend on previous work. Consequently, they cannot be launched this way unless the *LPlib* is told to wait until the needed data is available.

When launching a pipeline procedure, one may request the completion of a given list of previously launched procedures before actually running it. For this purpose, a table of dependency procedures can be supplied to the LaunchPipeline command.

Let's say procedures 1 and 2 are independent but procedure 3 needs some results from the first one :

```
main()
{
    ProcIndex1 = LaunchPipeline(LibIndex, procedure1, data1, 0, NULL);
    ProcIndex2 = LaunchPipeline(LibIndex, procedure2, data2, 0, NULL);

    TableOfIndex[0] = ProcIndex1;

    ProcIndex3 = LaunchPipeline(LibIndex, procedure3, data3, 1, TableOfIndex);
    ...
}
```

This way, procedures 1 and 2 will run concurrently on separate processors and, if 1 completes before 2, procedure 3 will even run concurrently with 2.

# 3 List of commands

## 3.1 AddDependency

### Syntax

```
AddDependency(LibIndex, Element1, Element2);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| Element1 | int | index of the element in the base table (type 1) |
| Element2 | int | index of the element in the dependency table (type 2) |

### Description

Make base table element1 dependent from secondary table element2. This is a one way dependency, if you need a reverse one, you just have to add the opposite link :

```
AddDependency(LibIndex, Element2, Element1);
```

### Example

Make every triangle dependent from their own nodes. The table TriTab[ i ][ 0 to 2 ] stores the three nodes of the triangle number $i$.

```
for(i=1; i<=NumberOfTriangles; i++);
    for(j=0; j<3; j++);
        AddDependency(LibIndex, i, TriTab[i][j]);
```

## 3.2 AddDependencyFast

### Syntax

```
AddDependencyFast(LibIndex, NmbType1, Type1Tab, NmbType2, Type2Tab);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| NmbType1 | int | number of elements in the base table (type 1) |
| Type1Tab | int * | table containing the base elements indices |
| NmbType2 | int | number of elements in the dependency table (type 2) |
| Type2Tab | int * | table containing the dependency elements indices |

### Description

This command works like *AddDependency*, but instead of adding a single dependency from one entity to another, it adds a set of dependencies: all elements in the table Type1Tab[] depend on all elements in table Type2Tab[]. It is useful when setting dependencies from an element like a tetrahedron to all its vertices or face neighbors.

## 3.3 BeginDependency

### Syntax

```
BeginDependency(LibIndex, type1, type2);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| type1 | int | index of base type whose elements will depend on those of type 2 |
| type2 | int | index of secondary type which will be referred to by the base elements |

### Description

Building a dependency between two kinds of elements is a three-step process :

1. initialize the dependency with BeginDependency : provide type 1 (base) and type 2 on which type 1 elements depend.
2. do as many calls to AddDependency as there are links between elements of type 1 and type 2.
3. stop adding new dependencies : EndDependency.

## 3.4 EndDependency

### Syntax

```
EndDependency(LibIndex, float StatTab[2]);
```

### Description

The ending call to EndDependency triggers the splitting of table 1 into blocks and the building of a compatibility matrix describing potential collisions between those blocks. If two type1 blocks share at least one common element of type2 (dependency type), then they cannot be assigned to concurrent threads.

Information about potential collisions is returned in the StatTab[]. The first scalar contains the average percentage of type1 elements pointing to type2 blocks. The second value gives the highest percentage among type1 blocks.

The lower those percentages are (3% or less), the easier it will be for the scheduler to find as many independent blocks as there are processors in order to achieve maximum efficiency. Otherwise, the parallelization factor returned by the LaunchParallel command may go down as low as 1 (no parallelism at all) in case every block are incompatible with each other.

This seemingly little detail is indeed of the utmost importance since it sets a program overall parallelism capability when working on indirect memory access loops.

Because the average number of block collisions is driven by the node connectivity and numbering, a proper renumbering algorithm must be applied to the whole mesh (nodes and elements altogether) prior to running the *LPlib*. Best known algorithms are Peano-Hilbert curve or "Z" curve renumbering (see R.Löhner's book [2] for an in-deep view of renumbering schemes).

Note that, without proper renumbering, the intrinsic connectivity featured by meshes coming from methods such as advancing front or Delaunay will prevent any parallelism beyond two threads. Conversely, most octree meshing software generate suitable numbering.

## 3.5 FreeType

### Syntax

```
FreeType(LibIndex, type);
```

### Description

Free memory used by this type data structures as well as the type index number. They may be reused by the next call to NewType.

## 3.6 GetDependencyStats

### Syntax

```
GetDependencyStats(LibIndex, int Type1, int Type2, float StatTab[2]);
```

### Description

Recall dependencies statistics from Type1 elements pointing to Type2 elements. It is useful when adding dependencies on the fly to check whether the collisions are low enough to allow for good parallelization speedup.

## 3.7 GetLplibInformation

### Syntax

```
GetLplibInformation(LibIndex, int *NumberOfProcessors, int *NumberOfAllocatedTypes);
```

### Description

Returns the number of threads the *LPlib* was initialized with and the number of data types that have been set up so far.

## 3.8 GetNumberOfCores

### Syntax

```
GetNumberOfCores();
```

### Description

This command simply returns the system's number of available cores.

## 3.9 GetWallClock

### Syntax

```
GetWallClock();
```

### Description

Returns a double that contains the physical time, the so-called "wall clock", in seconds.

## 3.10 HilbertRenumbering

### Syntax

```
HilbertRenumbering(LibIndex, NmbLin, box, crd, idx);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| NmbLin | int * | number of elements to be renumbered |
| box | double* | pointer to a table of 6 doubles containing the mesh bounding box |
| crd | double (*)[3] | pointer to the 3D coordinates table |
| idx | long (*)[2] | pointer to a table containing two entries for each renumbered item: crd[i][1] is the new index of item number i, crd[i][0] is the old index of item i |

### Description

See the sample code generate_hilbert_curve.c for more information.

## 3.11 HilbertRenumbering2D

### Syntax

```
HilbertRenumbering2D(LibIndex, NmbLin, box, crd, idx);
```

**Parameters**

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| NmbLin | int * | number of elements to be renumbered |
| box | double* | pointer to a table of 4 doubles containing the mesh bounding box |
| crd | double (*)[2] | pointer to the 2D coordinates table |
| idx | long (*)[2] | pointer to a table containing two entries for each renumbered item: crd[i][1] is the new index of item number i, crd[i][0] is the old index of item i |

**Description**

See the sample code generate_hilbert_curve.c for more information.

## 3.12 InitParallel

**Syntax**

```
LibIndex = InitParallel(NumberOfProcessors);
```

**Description**

Mandatory initialization of *LPlib* library prior to using any command. The sole parameter is the maximum number of processors to be used by the *LPlib*. This number may be greater than the computer's available processors for scalability testing purposes or lower in order to lighten system load (maximum = 128).

## 3.13 LaunchParallel

**Syntax**

```
acceleration = LaunchParallel(LibIndex, type1, type2, procedure, parameters);
```

**Parameters**

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| type1 | int | base type index over which to perform the loop |
| type2 | int | index of dependency type in case of indirect memory access loop, 0 otherwise |
| procedure | void * | pointer to a procedure that contains the parallelized loop |
| Parameters | void * | pointer to a single structure containing every parameter and data needed by the loop |

| Return | type | description |
|---|---|---|
| acceleration | float | number of average threads running together during the loop execution |

**Description**

In case of a simple direct memory access loop (without dependency type), the acceleration factor is equal to the number of processors.

If the loop has dependencies, the acceleration number will range from 1 (no parallelism because of too many interdependencies between blocks) and the number of processors (thanks to a perfect renumbering algorithm).

## 3.14 LaunchPipeline

**Syntax**

```
IndexOfProcedure = LaunchPipeline(LibIndex, procedure, parameters, SizeOfTable,
TableOfIndex);
```

**Parameters**

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| procedure | void * | pointer to the procedure to be launched in parallel |
| Parameters | void * | pointer to a single structure containing every data needed by the procedure |
| SizeOfTable | int | a number of procedures whose completion must be waited for |
| TableOfIndex | int * | indices of procedures to wait for before running this one |

| Return | type | description |
|---|---|---|
| index | int | identification index of this procedure |

### Description

The LaunchPipeline command allows for another kind of parallelism called pipelining. It is about launching a procedure without waiting for its completion (like the fork command in Unix).

If a procedure A launches procedures B and C in a row via LaunchPipeline, both B and C will start at once and will run concurrently with A.

If, say, C needs the completion of B before running, a dependency table may be provided when launching C to tell the scheduler to wait for B to complete before running C.

## 3.15 NewType

### Syntax

```
IndexOfType = NewType(LibIndex, NumberOfLines);
```

### Description

Defining a new table is easy : you just have to tell this procedure the number of lines contained in the table. It will return a unique index that should be provided to any procedure working on this table.

## 3.16 ParallelMemClear

### Syntax

```
code = ParallelMemClear(LibIndex, table, size);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| table | void * | pointer to a memory area that will be erased in parallel |
| size | long | number of bytes to be cleared |

| Return | type | description |
|---|---|---|
| code | int | error code is 1 if everything went right and 0 otherwise |

### Description

It works similarly to the C library memset command.

Parallel memclear is only useful for *ccNUMA* computers. It may not improve, or even degrade, speed for crossbar systems like most machines under 16 cores.

## 3.17 ParallelQsort

### Syntax

```
ParallelQsort(LibIndex, base, nel width, compar);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| base | void * | a pointer to data to be sorted in parallel |
| nel | size_t | number of elements to be sorted |
| width | size_t | size in bytes of an element |
| compar | int (*)(const void*, const void*)) | pointer to a function comparing two elements |

### Description

It works similarly to the C library qsort command.

## 3.18 ResizeType

### Syntax

```
IndexOfType = ResizeType(LibIndex, IndexOfType, NewNumberOfLines);
```

### Description

Increase the number of lines of a previously allocated data type. This command must be called outside of a running parallel loop and the NewNumberOfLines must respect this condition:

$$InitialNumberOfLines < NewNumberOfLines < 2 * InitialNumberOfLines$$

## 3.19 StopParallel

### Syntax

```
StopParallel(LibIndex);
```

### Description

Free all allocated memory and kill threads.

## 3.20 UpdateDependency

### Syntax

```
UpdateDependency(LibIndex, Type1, Element1, Type2, Element2);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| Type1 | int | type of the base element |
| Element1 | int | index of the base element |
| Type2 | int | type of the dependency element |
| Element2 | int | index of the dependency element |

### Description

Adds a new dependency to a previously defined dependency set. This command must not be called when a parallel loop is running. Contrary to the *AddDependecy* command, the elements kinds must be provided along with their indices.

## 3.21 UpdateDependencyFast

### Syntax

```
UpdateDependencyFast(LibIndex, Type1, NmbType1, Type1Tab, Type2, NmbType2, Type2Tab);
```

### Parameters

| Parameter | type | description |
|---|---|---|
| LibIndex | int | instance number of *LPlib* |
| Type1 | int | type of the base elements |
| NmbType1 | int | number of base elements |
| Type1Tab | int * | table of base elements |
| Type2 | int | type of the dependencies elements |
| NmbType2 | int | number of dependency elements |
| Type2Tab | int * | table of dependency elements |

### Description

Adds new dependencies to a previously defined dependency set. This command must not be called when a parallel loop is running. All Type1 elements in Type1Tab[] will depend on all Type2 elements in Type2Tab[].

### 3.22 WaitPipeline

**Syntax**

```
WaitPipeline(LibIndex);
```

**Description**

Setup a rendezvous point : this procedure will wait for every pipeline procedures to complete.

# 4 Glossary

## 4.1 A.P.I.

Application programming interface, a comprehensive list of all the functions provided by a library along with their arguments.

## 4.2 ccNUMA

Cache coherency nonuniform memory access. In such a system, memory requests travel through a network connecting local groups of processors called nodes. Even though throughput is lower and latency higher compared with a *crossbar* architecture, ccNUMA can expand to a greater number of processors (up to 8192 cores sharing the same memory vs. 256 for crossbar machines).

## 4.3 Crossbar

Memory interconnection matrix. In this architecture $n$ processors are connected to $n$ (more often $2n$ or $4n$ for efficiency reasons) *interleaved memory banks* through a connecting matrix providing each processor with its own dedicated data path. Consequently, throughput and latency are only slightly degraded compared with a single processor system. However, this kind of architecture can accommodate a limited number of cores (as of today, the biggest systems available features 128 dual-core processors).

## 4.4 Distributed memory

In such architecture, memory is not addressed as a whole, but located in independent units (such as a cluster computer). Those memory units (or spaces) should be addressed explicitly, making the parallelization of programs much more tedious. However, systems based on this model (mostly clusters) can expand to more than 100,000 processors.

## 4.5 Interleaved memory

Technic allowing concurrent accesses to a single memory image while keeping full performance. A memory space of size $s$ is split into $n$ chunks of size $s/n$ and addresses are spread across blocks through a modulo operator.

Lets say we have a 4-way interleaved 100 words memory : addresses 0, 4 and 80 fall in the same memory block 0 (or bank) because $0 \bmod 4 = 0$, $4 \bmod 4 = 0$ and $80 \bmod 4 = 0$. Likewise, addresses 1 and 17 fall in bank 1 and addresses 22 and 26 in bank 2.

Since each memory bank can serve a processor request concurrently with other banks, such a system can sustain= full access speed to each processor as long as their requests fall into different banks.

Whenever two or more processors request memory from the same bank, access will be serialized, thus greatly degrading the performance. This is why computer designers use two or four times more banks than cores, in order to lower the probability of such conflict.

## 4.6 *LPlib*

Acronym for Loop Parallelism Version 3, a fancy-less but explicit name.

## 4.7 Multi-core

A single chip processor hosting several cores.

The word processor may refer to a chip or a core inside a chip. It is important to distinguish between a chip (a single electronic component which may contain one or more processors) and a core (a processor located inside a chip). Consequently, a quad-processor computer could be made of four single-core chips, two dual-core chips or one quad-core chip.

## 4.8 Pthread

Posix thread, a lightweight process conforming to the posix norm.

A pthread is a task launched by a program and run concurrently.

In a multiprocessor system, a program may be running on a single processor and may launch several threads anytime. Those threads may be executed by other processors relieving the main program burden. That is what multi-thread programming is about.

## 4.9 Shared memory

A memory architecture where several processors share the same memory space. Such system facilitates the development of parallel applications since programmers don't have to care about their data localization among multiple spaces like in *distributed memory* systems.

However, those systems are more limited in number of processors and more expensive than their distributed memory counterparts.

# References


[1] B. NICHOLS, D. BUTTLAR & J. PROULX-FARRELL, Pthreads programming, *O'Reilly & Associates, Inc.*

[2] R. LÖHNER, Applied CFD Techniques, *WILEY.*

[3] SAGAN, Space-Filling Curves, *Springer Verlag, New York, 1994.*