Handling unstructured meshes in multithreaded environments with the help of Hilbert renumbering and dynamic scheduling

Loïc Maréchal

INRIA Saclay, France

Abstract

A software that deals with unstructured meshes is always tedious to parallelize because of its frequent indirect memory writes. These fine grain memory races are usually dealt with costly memory duplication and scatter-gather methods which require strong modifications in both algorithm and data structures. With the help of Hilbert renumbering and clever block dynamic scheduling, concurrent memory write problems can be addressed at a very low compute and memory overhead. Furthermore, such scheme always provides a very quick and easy parallelization of existing serial codes. It is, however, limited to shared memory machines but recent development in this area made tens of cores and terabytes of memory affordable, thus pushing the limit to billions of elements.

Keywords: unstructured mesh, multithreading, shared memory, write contention, Hilbert renumbering, dynamic scheduling

1. Motivation

Multicore processors have been around for ten years now, but the HPC community has not yet made the move toward it and is still sticking to distributed memory schemes like MPI [1]. Furthermore, huge shared memory machines now reach up to multiple Terra bytes and can handle tasks that were only accessible to clusters a few years ago.

From our own experience, working with researchers and engineers in the meshing and CFD fields, people are well acquainted with the problematic of distributed memory computing but feel very uneasy when it comes to shared-memory and multithreading.

Henceforth, we developed a library that can simplify the port of serial codes that deal with unstructured meshes as their main data. This is the case in our numerical simulation environment where the various levels of the software stack, preprocessing (automated meshing and adaptation), solving (finite element or finite volume methods) and post processing (visualization or solution interpolation), all presenting the same kind of memory access pattern and can be multithreaded with the same tools. So came the idea of the *LPlib* library that deals with these data structures.

The advantages over MPI are that the porting time is much shorter (days instead of months) and some acceleration can be achieved even if the code is only partially multithreaded while with distributed computing, a serial part may require a costly gathering of data that may spoil the whole speedup.

The main advantage over OpenMP [2], [3], and UPC [4] is its greater efficiency, thanks to the blocks decomposition that avoids costly memory duplication and renumbering that greatly enhances cache reuse.

Quick multithreading of existing serial codes. Assuming that most of the processing time of HPC software is spent on loops that run over elements or nodes, providing software developers with functions that automatically process these loops in parallel seems pretty obvious. The basic concept is to start with an existing serial code, give some information about the data structures, like links between elements and nodes, then move the loops into independent procedures so that they can be run in parallel by the *LPlib* library.

Step by step parallelization. Contrary to distributed computing, where the whole code needs to be modified so that it can access remote data, it is possible to mix sequential and parallel loops since the data structure does not change from the original serial code.

Hence, it is possible to start by parallelizing one loop as a test and then modify all other loops one by one depending on their importance in terms of execution time, or the programmer's availability.

I think that this step by step porting process is very useful for people who are reluctant to invest much time in parallelizing their software.

Little knowledge required. Most loops that deal with unstructured meshes presents the same set of characteristics and can be gathered into three kinds depending on the way they write to memory:

- independent memory writes: destination[i] = source[i]
- vector reduction: sum = sum + source[i]
- dependent memory write: dest[indirect[i]] = source[i]

All it takes to know is how to handle those three kinds of loops, no knowledge on hardware, distributed data structures or synchronization is required.

Low overhead, high efficiency. Since the library works on mesh partitions instead of single items and no user data is duplicated, the memory overhead and the scheduling time is kept very low, in the order of one to three percents.

Because no scatter gathers is needed and dynamic scheduling keeps synchronization needs very low, the efficiency is quite high on a single multicore chip.

Portability. The *LPlib* is based solely on the pthreads library that comes standard with all modern operating systems like Linux, Mac OS X and Windows (since Vista). Pthreads have been around for more than 20 years and there is no worry that they will disappear any time soon. Portability and longevity are two key points [5].

Competing solutions. Other ways to multithread a code exist like automatic parallelization features from some compilers (Intel, IBM), OpenMP or C language extension like UPC. All of them can deal easily with the first and second kind of loops, but they all fail to process concurrently loops that perform indirect memory writes, and those are very common when dealing with meshes.

In this situation, competing solutions require the user to manually duplicate the memory to prevent different threads from writing at the same location, then perform a final gather step, consequently losing memory and time.

2. Mesh renumbering

One of the main drawbacks of unstructured meshes is its non-linear memory access pattern. The most basic mesh structure is made of a nodes' coordinates table and an element's indices table that points on those nodes. If a loop is performed on consecutive elements and data associated with each element's nodes are accessed, these nodes are not consecutive in memory. Conversely if the loop is performed on consecutive nodes and data associated with each node's incident elements are accessed, the same random pattern is observed.

These gaps between indirect memory accesses trigger costly cache misses. Hence the need for renumbering the nodes and the elements so that indirect access loops present minimal gap between memory addresses, ideally making them consecutive, and when it is not possible, trying to reuse the data recently accessed so that it is still available in the cache memory.

2.1. Aspects of renumbering

There are numerous renumbering algorithms [6], [7], [8], [9], but in the scope of this paper, we will limit to the ones that are relevant to shared memory parallelism.

Cache coalescence. Processors do not read memory one word at a time but read a set of consecutive words, known as a cache line, in the hope that those extra words read may be accessed subsequently, reducing the need for further memory accesses. Presently, most processors use 64 to 256 bytes (or 8 to 32 double precision words) cache lines. Supposing a node uses four double words, it means that if two nodes from the same element are consecutive in the node table, accessing them will require only one cache miss instead of two, doubling the effective memory bandwidth.

Cache reuse. Since it is impossible to make every element's nodes consecutive when dealing with unstructured meshes, high speed may be achieved if a node has been recently accessed by a previous element and is still stored in the cache memory. Sizes of the various levels of caches vary greatly between processors, from tens of kilobytes for the fastest level one cache to tens of megabytes for slower level three cache.

Consequently, if an element's node has been accessed by one of the 1000 previous elements, chances are that it still resides in the first level cache and can be read or written without time penalty. Last level caches may hold up to a million nodes but is an order of magnitude slower than the first level one, which is anyway an order of magnitude faster than the main memory !

Sequential acceleration. The consequence of these two aspects is that a proper mesh renumbering may be beneficial to the running time even for sequential codes. Since main memory access is at least two orders of magnitude slower than the cache, substantial acceleration may be gained depending on the percentage of the execution time that is spent on memory accesses.

Parallel acceleration. Renumbering also dramatically affects parallel acceleration since the degree of concurrency reached by the code comes directly from the dependence between blocks of consecutive elements. If elements geometrically close to each other are also stored closely in memory, chances are that this mesh will exhibit high levels of concurrency as explained in a following paragraph (2.3).

2.2. Renumbering methods

The methods analyzed fall into two categories: advancing front and hierarchical.

Advancing front. Methods like Cuthill McKee [7] and Gibbs [8] fall under this category. They number entities starting from a seed and process by assigning consecutive numbers to neighboring entities. They produce numbering where the distance between elements' nodes is around $2^{2/3}$ the number of entities in the mesh, which is an average value. It is small enough so that cache reuse is good but nodes are usually too far away from each other to fit in the same cache line.

Hierarchical. Methods like Z curve [10], Peano [11] or Hilbert ([11] and [9]) subdivide the space recursively and set each entity number according to a predefined pattern (see Hilbert's curve in Figure 1). The numbering is quite different from the previous class: consecutive nodes have either close or very far index. Their cache reuse is excellent as well as their data independence which makes them more efficient than advancing front methods in terms of shared memory parallelism.

Comparison. Here are some statistics on five different methods when looping over tetrahedra and accessing their nodes indirectly.

- cache reuse: the chance that a given node belongs to the list of the previous 1000 nodes (ranges from 0% to 100%),
- cache coalescence: when accessing one of the tet's node, how many other nodes are read in one go (ranges from 1, no coalescence to 4, all four tet's nodes are consecutive)
- block dependencies: when the elements table is split into 512 consecutive chunks, what is the chance that any block has common nodes with the others (ranges from 0%, no collision and perfect parallelism, to 100%, no concurrency possible)

Method	Reuse	Coalescence	Dependencies
Delaunay ordering	8%	1.002	98.5%
Advancing front	82%	1.15	2.97%
Gibbs + C-McKee	75%	1.12	5.98%
Hilbert	84%	1.08	1.97%

The total speedup inherited from a renumbering is the product of it parallel speedup (the lower the block dependencies the better) and its sequential speedup (the higher the cache reuse and coalescence the better), hence the following acceleration formula:

$$acceleration = \frac{T_{sm} + T_{sc} + T_{pm} + T_{pc}}{\frac{T_{sm}}{C_r * C_c} + T_{sc} + \frac{\frac{T_{pm}}{C_r * C_c} + T_{pc}}{N * (1 - D)}}$$

Where:

- T_{sm}: time spent reading memory (cache miss) in the sequential part of the code,
- T_{sc}: time spent on computations in the sequential part of the code,
- T_{pm}: time spent reading memory (cache miss) in the parallelized part of the code,
- T_{pc}: time spent on computations in the parallelized part of the code,
- C_r: normalized cache reuse (this time ranging from 0 to 1),
- C_c: cache coalescence,
- N: number of threads used by the parallel section,
- D: normalized block dependencies, ranges from 0 to 1.

2.3. Implementation of Hilbert renumbering

Dependencies matrix. One of the key points in the implementation of the library is the way it partitions the mesh in order to reduce the time and memory needed to make sure that no elements sharing a common Verex are processed concurrently. To do so, the *LPlib* stores dependency information on *super entities* instead of real entities.

The partitioning is trivial, a table of N_{item} is represented by $64*N_{threads}$ super entities that represent each $N_{item}/(64*N_{threads})$ single entity. Each super entity represents a set of consecutive single entities, so for example if we have $1,000,000$ triangles to process with four threads, the number of super triangles will be $4 * 64 = 256$ and each super block will represent $1,000,000/256 = 3906$ triangles.

Accordingly, super triangle number 1 will stand for triangles ranging from 1 to 3906, super triangle 2 for triangles 3907 to 7812, and so on. The same will be applied to Verices, represented by super Verices so that, when the user sets dependencies between $1,000,000$ triangles and $500,000$ Verices, the library will internally build a $256*256$ dependency matrix whose lines represent the super triangles and columns represent the super Verices.

If the matrix reads $m[5,2] = 1$, it means that super triangle 5 uses super Verex 2, so while a thread is processing this super triangle, no other threads should process a super triangle that uses the super Verex 2. Checking the compatibility between two super elements is straight forward: their respective matrix lines must not have common values set to one.

Keep in mind that a super element represents a range of single elements, so when the matrix reads $m[5,2] = 1$, it means that at least one triangle ranging from $19,531$ $(5 * 3906 + 1)$ to 23436 $(6 * 3906)$ has a Verex ranging from 3907 $(2 * 1953 + 1)$ to 5859 $(3 * 1953)$, so there may be false sharing.

If the mesh elements and Verices are numbered in a random way, as it is the case when generated by a Delaunay method, chances are that the matrix will be almost filled with 1. Let's say that triangle 1 is made of Verices $1,000$, $6,000$ and $13,000$, this triangle belongs to super triangle 1 $(1 + IP(1/3906))$ and touches super Verices 1, 4 and 7. So this single triangle will set to 1 three columns of the matrix line corresponding to the super triangle 1. Since triangles ranging from 1 to 3906 all affect the same line, if their Verices were randomly numbered, they would probably span all the range from 1 to $500,000$, thus filling all the columns. See Figure 2 on the left.

In the end, the number of non-null values in the matrix will define the number of blocks that can be processed concurrently. The sparser the matrix, the higher the parallel scaling.

In order to get a sparse dependency matrix, the mesh needs to be renumbered with a Space Filing Curve like Hilbert or Peano. Since this paper does not aim at presenting the whole SFC theory, readers should check [11] and [9] for more information.

Fast parallel Hilbert renumbering. The *LPlib* features a fast Hilbert renumbering that has been implemented with fast logical operation and bit shifts on integers instead of slower floating point arithmetic. It is, of course, multithreaded.

On a quad core i7 laptop, it renumbers 2.5 million elements per second.

Low block conflicts and high cache reuse. Not only does the Hilbert renumbering dramatically reduce block conflicts (Figure 2), but it also greatly enhances cache line reuse and eventually speeds up the serial code as shown in the table ???.

Even if the blocks are not perfectly compact and the interface between them is not minimal, we do not care, since all we want is to have a wide choice of independent blocks to choose from. The partitioning requirements are much looser than those of distributed computing where minimizing interfaces is critical.

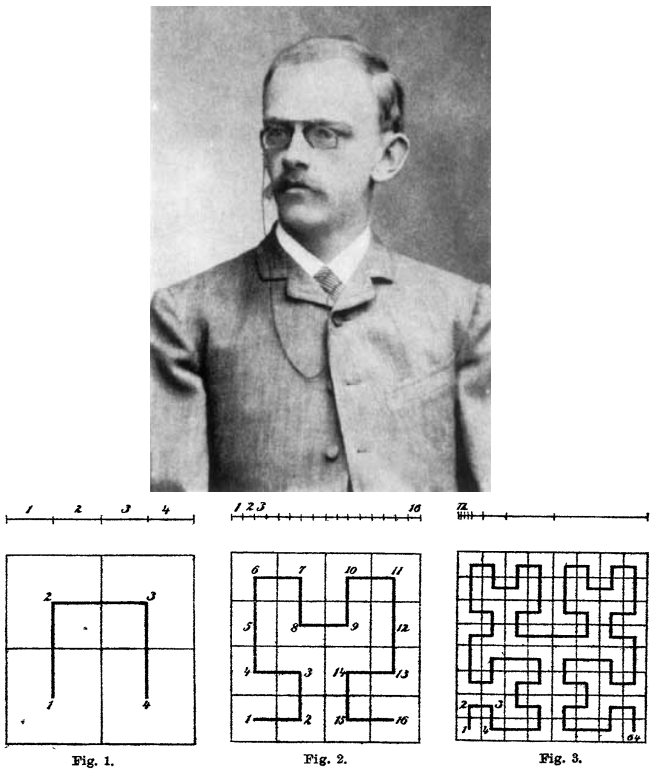

Figure 1: German mathematician David Hilbert in 1886 and his renumbering scheme as published in his great two-page paper [9] in 1891.

3. Dynamic scheduling

When the library is first initialized with a given number of processes, it launches as many computing threads that sit idle, waiting to be woken up by the master thread for some work to do. When a *LaunchParallel* command is issued, the master thread wakes-up the computing threads one by one, signaling each that they should search for some work to do. Note that the master thread is only sending commands and is not scheduling

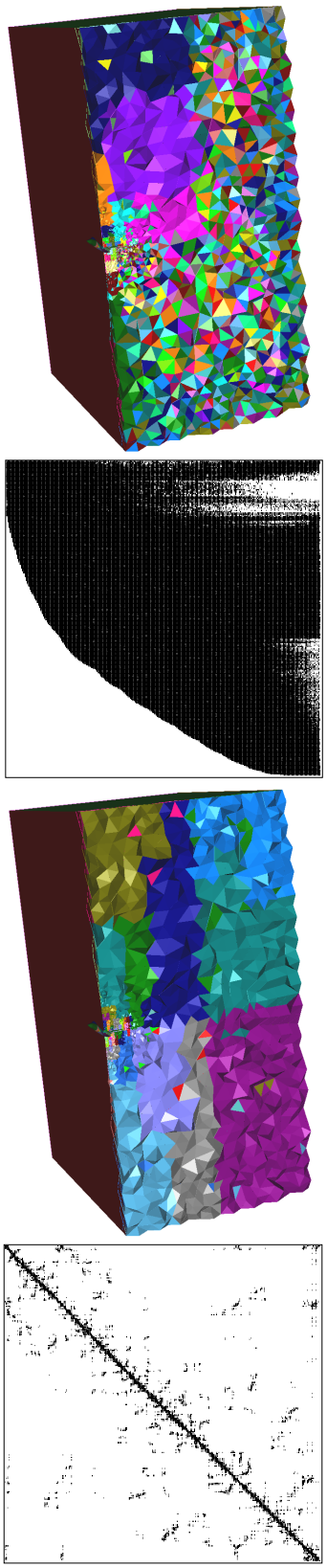

Figure 2: Volume mesh around an F15 aircraft (above), and the corresponding dependency matrices (below). On the left is the original number right from the tetmesh GHS3D and on the right is the Hilbert renumbered version.

the jobs, the scheduler is called by each thread as soon as they wake-up.

As for now, only one thread may call the scheduler at a time, which is the bottleneck of this method. That is why the scheduler has to be as fast as possible. It searches through the list of uncompleted blocks to be processed and check whether their dependency word (the list of super Verices they use) are compatible with that of the other threads' blocks that are presently being processed.

Once it finds a suitable job, the thread starts looping over the block's range of elements. If there were still some blocks to be processed, but none was compatible with those running, the thread enters waiting mode and will be awakened only when another thread completes its job.

At this point, it may be possible for both threads to find compatible jobs. This situation occurs near the end when almost every block has been processed and the scheduler has very little choice left. To mitigate this problem, blocks that have the most bits set to 1 in their dependency word, which are the more constrained, are treated first.

When all the blocks have been processed, the threads enter idle mode until a new parallel loop is launched. Figure 4 illustrates the process with four threads.

Dependency word. At this point we have a $256 * 256$ matrix that is filled with 0 or 1 so 32 columns can be packed inside an integer value to save storage and increase processing speed. Each line of the matrix is made of 256 bits, or 8 integers, that I call the *dependency word*. In order for two blocks to be compatible, they must not have any common column value set to 1.

This is easily checked with the logical *AND* operator.

Bitwise operations on words: and, or, xor. Performing the various operations on dependency words is really fast since it involves only logical operations, or, exclusive or, and, which are the fastest on a processor. Performing these operations on a table of 8 consecutive integers can be done in parallel with Intel SSE or AVX vector extension. AVX2 is even capable of performing 8 integers OR, XOR or AND in one single machine instruction which make the scheduling really fast.

Searching for a job. Consists in looping over uncompleted blocks and performing a logical AND with each block's dependency word and the global running dependency word. If the answer is true, it means they have at least one bit in common, so they may share a common Verex and as a result, they cannot be processed concurrently.

If false, the tested block is compatible with the other running blocks and can be processed concurrently.

Adding a new job. When a new job is found, its dependency word must be added to the global running dependency word. This is a trick to speed things up. We could simply have stored a list of running blocks' index and when searching for a new block to process, checked its dependency word with all running blocks' words.

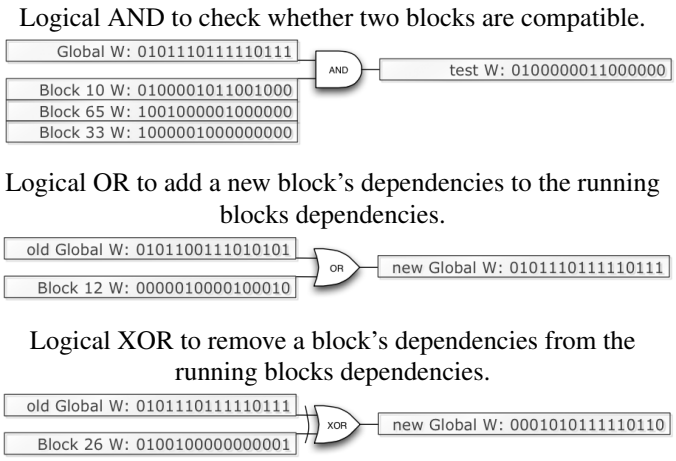

Figure 3: Bitwise operations used on multibyte words by the scheduler.

Since the running blocks are independent from each other, their words have no common bits by definition, so we can safely perform a logical or between all of them and store the result in a global running dependency word.

When searching for a new block, all we have to do is computing the logical and with the candidate block's word and the running word.

Removing a completed job. Likewise, when a job has completed, we need to remove its dependency information from the global running word in order to remove useless constraints. This is very easily done by computing the exclusive OR between the completed block's word and the running word and store the result in the latter.

Since the completed block has no bits in common with the others running blocks words, there is no risk for a bit to be cleared while the corresponding super Verex is still being processed by another thread.

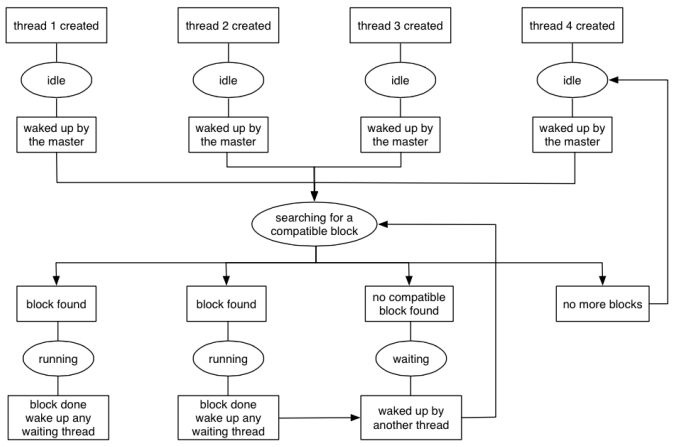

Figure 4: Sample run: launch, wait, search, run, blocked, wake-up, end

Robustness: multiuser environment, asymmetric workloads, dynamic dependencies. Such approach presents many advantages.

First, it is very independent from workload imbalances since there are 64 times more blocks than processing threads, if some block takes more time to be processed for some calculation reasons (boundary conditions, slow local convergence, higher cache miss, slower distance memory accesses), the other threads do not have to wait for the slower block to complete and may process several other blocks in the meantime.

Furthermore, if the machine is used by other users or some system tasks starts perturbing the parallel code, the core running one of the *LPlib*'s threads may be assigned to some other tasks by the system's scheduler and the block being processed will take much longer. As seen above, it is not a problem for a dynamic scheduling scheme since the remaining blocks will be processed by the other available cores.

4. *LPlib*: a loop multithreading library

As stated before, the *LPlib* is a loop parallelization scheme, all it takes is to tell the library how many entities of different kinds the mesh is made of (Verices, triangles, hexahedra, etc.), and for each loop, on which entity they run over.

This project is open-source and freely available on GitHub [12].

Data types. A data type is a generic concept that is made of two scalar values, one that is the entity tag, to differentiate it from the others, and the number of items it is made of.

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.

4.1. Example 1: computing the mean value for each triangle of a mesh

Let's say we have a mesh stored in a structure of type MeshStruct, comprising a table Ver of nbv Verices and a table Tri of nbt triangles.

```
typedef struct {
  double t; // temperature
  int    d; // Vertex degree
}VerStruct;

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

typedef struct {
  // global maximum value
  double max;
  // table of local maximums
  double lMax[ NmbThreads ];
  int nbv, nbt;
  VerStruct Ver[ nbv ];
  TriStruct Tri[ nbt ];
}MeshStruct;
```

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

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

Serial code.

```
main() {
 for(i=0;i<msh->nbt;i++){
  msh->Tri[i]->t = 0;
   for(j=0;j<3;j++)
    msh->Tri[i]->t += msh->Tri[i]->Ver[j]->t/3;
  }
}
```

Parallel code.

```
main() {
 // Initialize LPlib with 4 threads,
 // declare nbt triangles
 // and launch the parallel loop
 iLib = InitParallel(4);
 tTri = NewType(iLib, msh->nbt);
 LaunchParallel(iLib,tTri,0,ComputeAverage,msh);
 StopParallel(iLib);
}

ComputeAverage(int begin, int end,
               int iThread, void *arguments){
 // Fetch all arguments from a single structure
 MeshStruct *msh = (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++){
  msh->Tri[i]->t = 0;
   for(j=0;j<3;j++)
    msh->Tri[i]->t += msh->Tri[i]->Ver[j]->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, msh);
ComputeAverage(250, 499, 1, msh);
ComputeAverage(500, 749, 2, msh);
ComputeAverage(750, 999, 3, msh);
```

4.2. Example 2 : searching for the maximum value among a set of triangles

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

Serial code.

```
main() {
 msh->max = -273.15;
  for(i=0;i<msh->nbt;i++)
   if(msh->Tri[i]->t > msh->max)
    msh->max = msh->Tri[i]->t;
}
```

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

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

Parallel code.

```
main(){
 // Launch the parallel part of the code
 iLib = InitParallel(4);
 tTri = NewType(iLib,msh->nbt);
 LaunchParallel(iLib,tTri,0,ComputeMaximum,msh);
 StopParallel(iLib);

 // Perform a loop on the table of local
 // average values computed by each thread
 // in order to find the global maximum value
 msh->max = -273.15;
 for(i=0;i<4;i++)
  if(msh->lMax[i] > msh->max)
   msh->max = msh->lMax[i];
}

ComputeMaximum(int begin, int end,
               int iThread, void *arguments){
 MeshStruct *msh = (MeshStruct *)arguments;

 // The local maximum value is stored in
 // the maximums' table under the index
 // given by the thread number
 msh->lMax[ iThread ] = -273.15;
  for(i=begin; i<end; i++)
   if(msh->Tri[i]->t > msh->lMax[ iThread ])
    msh->lMax[ iThread ] = msh->Tri[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:

```
ComputeMaximum(int begin, int end,
               int iThread, void *arguments) {
 MeshStruct *msh = (MeshStruct *)arguments;
 double thread_max = msh->lMax[ iThread ];

 // The local maximum value is stored
 // in the maximums' table under the
 // index given by the thread number
 msh->lMax[ iThread ] = -273.15;
  for(i=begin; i<end; i++)
   if(msh->Tri[i]->t > thread_max)
    thread_max = msh->Tri[i]->t;
 msh->lMax[ iThread ] = thread_max;
}
```

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 for 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 accesses 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.

4.3. Example 3: an indirect memory access loop featuring a cross dependency problem

Using meshes and matrices in scientific software leads inevitably to more complex algorithms where indirect memory accesses are needed. For example, accessing a Verex structure through a triangle link in the following instruction:

```
Tri[i]->Ver[j];
```

Such memory access is very common in C, the compiler will first look for a triangle *i*, then a Verex *j*, thus accessing the data indirectly. If a loop is done over triangles, adding their temperature to their three Verices' own temperature value, it would lead to a writing conflict.

Serial code.

```
main() {
 // main loop adding the triangle's temperature
 // to each of its Verices temperature.
 // The added value is divided by the Verex'
```

```
 // degree to compute a mean value.
 // The degree is the number of triangles
 // sharing the same Verex.
 for(i=0;i<msh->nbt;i++){
  tri = msh->Tri[i];
   for(j=0;j<3;j++)
    tri->Ver[j]->t += tri->t / tri->Ver[j]->d;
}
```

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

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

Even though there are no two blocks sharing the same triangles, it is not the case when it comes to Verices. For example, triangle 250 from block 1 and triangle 750 form block 3 may share the same Verex. If two threads where to process those triangles at the same time, they could write different temperatures at the same memory location, resulting in an undetermined value.

To properly handle this situation, the *LPlib* must be provided with the mesh connectivity. That is the list of Verices pointed to by triangles. This way, the library is able to split the triangles table into a certain number of chunks (typically 64 blocks per thread), and compute the dependencies between them.

The *LPlib* will provide each thread with blocks that not only do not share common triangles, but whose triangles do not share any Verices. Only this way all threads could safely run together.

Parallel code.

```
main() {
 // Init LPlib and define triangles
 // and Verices datatypes
 iLib = InitParallel(4);
 tTri = NewType(iLib, msh->nbt);
 tVer = NewType(iLib, msh->nbv);

 // Set the dependencies between
 // triangles and Verices
 BeginDependency(iLib, tTri, tVer);
 for(i=0;i<msh->nbt;i++)
  for(j=0;j<3;j++)
   AddDependency(iLib,i,msh->Tri[i]->Ver[j]->num);
 EndDependency(iLib);

 // Launch the parallel loop
 LaunchParallel(iLib,tTri,tVer,AddTemp,msh);
 StopParallel(iLib);
}

// Loop adding a triangle's temperature
// to its Verices in parallel
AddTemp(int begin,int end,
        int iThread, void *arguments){
 MeshStruct *msh = (MeshStruct *)arguments;
 for(i=begin; i<end; i++)
  for(j=0;j<3;j++)
   msh->Tri[i]->Ver[j]->t +=
     msh->Tri[i]->t / msh->Tri[i]->Ver[j]->d;
}
```

4.4. Dynamic data structures

Usually solvers use static data, so initializing the data types and the dependencies needs to be done just once before the calculation starts. But when it comes to mesh generation or modification, the number of entities may change (adding or removing elements), or some entities may be modified (like swapping and edge between two adjacent triangles).

In this case there is no need to rebuild all the data initialization but simply to update them with the help of *ResizeType* and *AddDependency* commands. Doing so might degrade the Hilbert renumbering properties (see bellow) and after strong mesh modifications, it might be useful to free all data types and restart the whole initialization and dependency process.

5. Use cases

The library has been used by in house projects since 2008 and various codes have thus far been multithreaded like a hex mesher, a flow solver and a mesh visualizer. This section will present information about the porting process like the developing time required, the percentage of each codes that have been multithreaded and the speedups we got on various hardware.

Meshing software: Hexotic, Optet. As the author of the library I first tried it on two of my meshing programs, Hexotic, an automated hexahedral mesh generator ([13]), and Optet, a tetrahedral mesh quality optimizer.

In Hexotic, only the final node smoothing and some metric calculation representing 70% of the total CPU time have been parallelized. Consequently, the Amdahl's serial factor is 0.25 and so the maximum theoretical speedup is no more than 4.

With Optet, all the loops, tetrahedra neighbors links, edge swapping and node smoothing have been parallelized and only the input-output remains sequential. Its Amdahl serial factor is 0.04 and the maximum speedup is 25.

Solver: Wolf. 2D and 3D Mixed-Element-Volume solvers for compressible Euler and Navier Stokes equations ([14]).

Post processing: metrix, interpol, vizir. Vizir: a light, simple and interactive mesh visualization software, which handles curved meshes, high order elements and solutions, hybrid elements (pyramids, prisms, hexahedra) and solutions visualization (clip planes, capping, iso-lines, iso-surfaces). It is freely available at vizir.inria.fr Interpol: fields transfer or interpolation between meshes in 2D and 3D. Metrix: compute 2D and 3D metric field based on a priori error estimate.

Multithreaded software. Some facts on codes parallelized with the library.

Program	Hexotic [13]	Interpol [15]	AMG+Metrix [15]
Description	hexahedral mesh generator	solution interpolation	metric field smoother & mesh adaptation
Lines of code	20,000	54,000	73,000
Porting time	1 week	1 week	1 week
Amdahl factor	0.25	0.10	0.06
6-core speedup	2.7	4.0	4.6
12-core speedup	2.7	5.4	7.3
20-core speedup	3.2	6.0	9.5
40-core speedup	3.8	8.6	N/A

Program	Optet	P1toPk	Wolf [16]
Description	tetrahedral mesh optimizer	high order meshing	flow solver
Lines of code	7,000	6,000	250,000
Porting time	2 weeks	1 day	2 weeks
Amdahl factor	0.08	0.007	0.02
6-core speedup	4.1	5.6	5.7
12-core speedup	6.0	10.7	10.7
20-core speedup	8.8	17.8	14.7
40-core speedup	9.3	36.6	20.2

It is important to note that the speedup is only dependent on the number of elements in the mesh. It takes roughly a thousand entities per core to get the best out of multithreading. Otherwise, the scheduling takes longer than the benefits of parallelism.

There is a complete independence from the geometry, mesh structure, local refinement, elements anisotropy, boundary condition or any physic related inputs. This is an important advantage over distributed models whose scaling is generally dependent on all those aspects.

Influence of multiple chip systems. The most basic multicore system is made of one chip containing multiple cores. But some systems go beyond that by using multiple chips, each with multiple cores. Each chip controls part of the memory in a round-robin way in order to mitigate the memory placement unbalance, so for example, in a dual chip system, even memory pages are handled by the first chip, and odd ones by the second chip.

Indeed, the *LPlib* has no concept of memory locality and is not *NUMA aware*, so it is best to set the NUMA default policy to interleaved memory across all nodes. These systems are called distributed shared memory and are a hybrid kind between purely distributed systems like clusters and true shared memory ones that are single chip. To get the best out of those multi-chip systems, a program should make sure that the data a chip's core is processing is located on the local chip's memory and stays there during the whole calculation, which is no trivial task.

The *LPlib* aims only at taking advantage of single chip shared memory architecture and dealing with distributed shared memory is beyond its scope. Anyway, on dual chip systems the scaling is still pretty decent and the performance stalls at quad chips and above.

Influence of dynamic over-clocking. The latest Intel CPU do not have fixed frequency but modify it between a wide range depending on the chip's temperature. When running a code sequentially, the chip runs cooler and the controller is able to raise the clock near its maximal value. On the other hand, when all cores are being used by a perfectly parallelized software, the clock goes down at its lowest bound.

So, on a quad core i7 laptop, we are comparing a single core running at 3.7 GHz against 4 cores running at 2.7 GHz, the game is unfair and it spoils the scaling ratios.

Influence of the number of cores over the number of memory channels. Another factor of great influence to take into account is the number of memory channels that connect the cores to the local memory. When choosing a computer, people usually tend to pay attention only to the number of cores per chip and not to the number memory channels which is much lower, and not adVerised by the manufacturer.

Codes that perform a lot of calculation for each memory read, like meshers and finite volume solvers, can accommodate a 2 to 1 ratio between the number of cores and channels. But a software that is memory constrained like in linear algebra needs a 1 to 1 ratio, so their practical speedups are bound the number of memory channels regardless of the number of cores.

It is important to note that a chip's complexity grows linearly with the number of cores, but with the square of the number of memory channels. It explains why AMD, IBM and Intel are all stuck with four channels while the number of cores can reach up to 18.

In the end, this problem will severely limit the room for improvement of shared memory parallelism.

6. Conclusion and further work

Real efficiency of distributed memory codes. The theoretical scaling capability of a cluster is, of course, orders of magnitude higher than that of a multicore chip. But as programmers using MPI know too well, if hard work is guaranteed, the scaling is not !

We have always wondered what is the scaling of real life applications on real life data ?

A partial answer came from Intel who gave a very interesting insight in their [17], where it appears that many applications scale between one or two orders of magnitude, which is good but not great.

In this regard, the single order of magnitude gained by multithreading is far from being ridiculous, especially when development time is considered, which is orders of magnitude in favor of shared memory.

Big data. Another point to mention is the dramatic increase in shared memory sizes over the last couple of years.

A terabyte of memory is now a common option on Intel Xeon E5/E7 servers and costs around $10,000, which is affordable even for the small team I work with. High-end X86 servers now reach up to 12 TB and, as they still cost a pretty penny as of 2015, they will become much cheaper in the coming years as demand for *big data* grows.

With such machines (we have a 40 cores, 1 TB memory server in-house) we are able to generate 1 billion hexahedra and 3 billion tetrahedra, mesh sizes that were previously only manageable on clusters.

Ease of use. In conclusion, the *LPlib* offers the HPC programmer an alternative to the MPI/distributed approach that is worthy of consideration if the goal is simply to process one million to one billion elements meshes in a reasonable time. Above one billion elements, of course, distributed memory is the way to go.

Future works: parallel stacks, parallel scheduler, hybrid parallelism.

- The loop is not the only iterator found in numerical simulation, a lot of work is also based on FIFO or LIFO stacks. It could be possible to develop a parallel stack iterator based on the basic block decomposition used in the *LPlib*. A *super stack* could be designed and whole blocks of elements independent from each other could be pushed or popped. It would not be as efficient as a fine grain stack because, if one element is pushed on the stack, the whole block would be pushed along. Consequently, tt may require smaller blocks, at the cost of slower scheduling.

- As for now, the scheduler is a critical section that only one thread may call at a time. We experienced that, when running on a SGI UV 1000 with 160 cores, it becomes a bottleneck. Scheduling could be parallelized by splitting the main block list into smaller lists and by accessing these lists in a circular way. For example, thread number i may search through available blocks from the list, number i, and on a second round, thread number i may look for a job in list number $i + 1$.

- One obvious last idea is to hybrid distributed and shared memory. It could help address some of the scaling problems that many distributed memory codes have when running on more than a thousand cores. Since the issues they face are different from that of shared memory models, it might be a good idea to run $\sqrt{n}$ MPI processes that would launch $\sqrt{n}$ threads on each node. It would be very interesting to see what kind of performance could be obtained on the latest 100 Gbit/s, 36 ports infiniband switches connecting 36 core nodes made of dual 18 core Xeon E2699-v3 (hence, a perfect 36*36 system).

References

[1] R. Ramamurti, R. Löhner, A parallel implicit incompressible flow solver using unstructured meshes, Computers & Fluids 25 (2) (1996) 119 – 132.

[2] OpenMP Application Program Interface (2013).
URL `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`

[3] Y. Sato, T. Hino, K. Ohashi, Parallelization of an unstructured navier–stokes solver using a multi-color ordering method for openmp, Computers & Fluids 88 (2013) 496 – 509.

[4] G. F. Dan Bonachea, Upc language specifications, version 1.3, Tech. rep., LBNL and UC Berkeley (2013).
URL `http://upc.lbl.gov/publications/upc-spec-1.3.pdf`

[5] D. B. . J. P.-F. B. NICHOLS, Pthreads programming, O'Reilly Media, 1996.

[6] SOME USEFUL STRATEGIES FOR UNSTRUCTURED EDGE-BASED SOLVERS ON SHARED MEMORY MACHINES.

[7] Reducing the bandwidth of sparse symmetric matrices.

[8] W. N.E.GIBBS, P.K.STOCKMEYER, An algorithm for reducing the bandwidth and profile of a sparse matrix, SIAM J. Numer. Anal., Vol. 13, Pages 236-250 (1976).

[9] D. Hilbert, Über die stetige abbildung einer linie auf ein flächenstück, Mathematische Annalen 38 (1891) 459–460.

[10] R. LÖHNER, Applied CFD Techniques, WILEY, 2008.

[11] Sagan, Space-filling curves, Springer Verlag, New York (1994).

[12] L. Maréchal, A parallelization framework for numerical simulation (2018).
URL `https://github.com/LoicMarechal/LPlib`

[13] Advances in octree-based all-hexahedral mesh generation: handling sharp features.

[14] F. Alauzet, A parallel matrix-free conservative solution interpolation on unstructured tetrahedral meshes, Comput. Methods Appl. Mech. Engrg.Submitted (2015).

[15] F. Alauzet, A. Loseille, A decade of progress on anisotropic mesh adaptation for computational fluid dynamics, Computer-Aided Design 72 (2016) 13 – 39, 23rd International Meshing Roundtable Special Issue: Advances in Mesh Generation. doi:https://doi.org/10.1016/j.cad.2015.09.005.

[16] F. Alauzet, A. Loseille, High-order sonic boom modeling based on adaptive methods, Journal of Computational Physics 229 (3) (2010) 561 – 593. doi:https://doi.org/10.1016/j.jcp.2009.09.020.

[17] B. Y. Wolfgang Gentzsch, The ubercloud hpc experiment: Compendium of case studies, Tech. rep., Intel (2013).