A run time support for the Divide and Conquer pattern

Emanuele Vespa, Lorenzo Anardu

August 27, 2010

Abstract

We focus in this work on the theoretical and practical aspects regarding the design and the realization of a run time support for the divide and conquer based computations. We will first derive a strong conceptual model for the divide and conquer parallel pattern, and furthermore we will give a possible working implementation of a run time support for that computational model. The motivation for this work are the heavy computations done by divide and conquer algorithms on large sets of data, which are characterized by the replication of same operations on different data, allowing so a very efficient parallel implementation. Our implementation will be able to exploit both single machines with multiple core/processors and homogeneous clusters architectures.

1 The theory

In this section will be given an overview of the theory that stands behind our work. Furthermore instead of referring to a concrete architecture (multi-core, multi-processor or clusters), we will use generic processing entities, in order to treat from a general point of view these topics.

1.1 The Divide and Conquer strategy

Definition 1. *The divide and conquer strategy consists on breaking the original problem in sub-problems of smaller size, solve them recursively, and combine these intermediate solutions in order to obtain the final solution.*

We can identify three basic steps involved in each level of recursion:

Divide : the problem is broken into a number of sub-problems.

Base case computation : after the recursion stops, solve the each sub-problem.

Combine : combine the solution to the sub-problems in order to obtain the final solution.

Hence we can give a sketch of a generic divide and conquer algorithm :

```
function divide_and_conquer(problem){

        if basecase ( problem) then
        return baseSolve( problem )
        else
        [problem_1 , problem_2 , ... , problem_n] = divide(problem)
        return combine( divide_and_conquer( problem_1 ) ,
            divide_and_conquer( problem_2 ), ... ,
            divide_and_conquer( problem_n ))
}
```

This kind problem solving technique has been recognized as a important source of both control and data parallelism. In fact it appears quite natural as divide and conquer algorithms can be parallelized: we can in principle execute every step of recursion in parallel. Through the divide phase we can divide the problem in smaller sub-problems and solve them in parallel, obtaining so a control parallelism at the task level. Some consideration can be done on the specific divide and conquer algorithm: for instance consider two well known sorting algorithms, the mergesort and the quicksort. Both use the divide and conquer strategy in order to sort an input data, but they are very different in the divide and combine phase. The mergesort split the data in a *balanced* way and operate on the data during the base solving and combine phase. The quicksort instead perform the sorting during the divide phase, and in this way it generates sub-problems of different sizes, leading to an *unbalanced* computation. Clearly, we can notice that, in the case of mergesort, the number of recursive steps is fixed for a given problem size, on the contrary in the quicksort this is not true since the number of sub-problems is data dependent. All of this observations can give an idea of how difficult can be to obtain a general cost model for this kind of computations. In the next sections, we will try to give a detailed documentation on two possible way of deal with the parallelization of divide and conquer algorithms: the nested parallelism approach and the map approach.

1.2 The recursive approach

Definition 2. *The nested parallelism is an extension of the flat control or data parallelism that permit to nest statically or dynamically additional levels of parallelism inside the original parallel tasks.*

The recursive approach, and so exploiting the nested parallelism, seems to be the natural way in which the divide and conquer algorithms can be parallalelized. If every task is capable of spawn other parallel tasks, hence to add levels of parallelism , then we can imagine that our algorithm can be parallelized simply following the recursion tree. We can choose a root node that will start the algorithm, and according to the divide function, will split the input and distribute it to the other nodes, which in turn, will recursively split and distribute the chunk of input data received. After the base case will be reached, then every node will sends back to its father the chunk of data computed, following from the leaves to the root the recursion tree. Through this approach we can reach both control parallelism and data parallelism. Since we are not interested in stream of data, but in computing a single input data, we can assert that through the divide and conquer strategy we can exploit parallelism at data level, by the fact that various chunk of the same input data are computed in parallel. In general we have also control parallelism, since more different task can be computed in parallel, e.g. in the quicksort it's probable that nodes on the left or right hand of the recursion tree are already performing the combines, meanwhile others are still performing combines.

1.3 The map approach

Definition 3. *In a data parallel computation, the input data are partitioned between workers, which execute the same function on their own partition of the input data.*

A data parallel computation can be of map type, where every worker operates both in reading and writing mode just on his partition, without the necessity of communications between workers. In a map computation we can devise tree phases: the scatter phase, the map computation phase and the gather phase. Through the scatter function we distribute the partitions of the input data (from now on they will be called chunks) to the workers. The map computation is the actual function applied by the workers to the chunks received. Finally in the gather phase the original data structure is reconstructed. We can try to exploit this kind of computational model for our purposes, adapting the divide and conquer strategy to the map computation. In order to do so, we have to modify the criteria by which the input data is partitioned and how is scattered to the processing entities. We can first notice that in the map, the partitioning of the data does not involve particular computation: this is not our case since in divide and conquer algorithms the data are divided according to a certain strategy. For example, in the mergesort algorithm the data are simply divided by a factor of n, on the contrary in the quicksort the data are divided according to the sort order, hence we have that the actual computation is done during the partitioning of the data. The question that arise now is: how can we combine this two models? The most intuitive way to do so is to think about a map model with the scatter and gather phases driven by the divide and combine strategy of our divide and conquer algorithm. This can be done as it follows: a first node (namely node zero) read the input data and execute the divide computation until it reaches a certain number of sub-tasks. At this point node zero scatter chunks to the processing entities, which in parallel execute *sequentially* the divide and conquer algorithm. As a processing entity finish its work, it sends back to node zero the result. As soon as node zero has received back all the scattered chunks, it can proceed with the final combines, according to the combine procedure specified by the algorithm.

Now that the two approach have been defined, we can proceed on a comparison between them.

1.4 Recursive versus Map approach

As said before, the nested parallelism is the most natural way to express parallel divide and conquer algorithms. But we have to face some serious problems from a formal point of view in order to give the user a detailed performance cost model. In this model, the number of communication between processing entities could be very high, so we need to find a threshold in order to stop the spawn of more parallel tasks. This because if the computation grain is fine, then the overhead for the communication is too high, leading to a very bad efficiency and scalability. This threshold is usually expressed in function of the division degree, the communication time and the input data size, but on the contrary of what happens in the map, it's not so easy to estimate. In principle, we can keep the threshold equal to the reaching of the base case, having in this way an amount of parallel tasks equal to the nodes of the recursion tree: this can affect very badly the performances, except in the case that the computational grain is very coarse. A map implementation of the divide and conquer gives us some great advantages from the cost model point of view. In fact it is well known the cost model for this kind of computation and if we succeed on combining this aspect with the flexibility of the division and the combine phase of the divide and conquer algorithms we can achieve important improvements on speed-up and efficiency. Even from an implementation point of view is simpler to deal with the map approach, since we can avoid deadlocks and complex synchronization procedures. Since with the nested parallelism is required that every node both waits for input data and sends output data in the same time, a careful design of the communication model is required in order to avoid deadlocks. Another unavoidable problem is the handling of *unbalanced* computations. For the nested parallelism we can use two different strategy according to the type of computation: in case of balanced algorithms we can schedule workers in a circular way starting from a node randomly chosen, for unbalanced algorithm it is better to pick workers completely random instead. In case of map approach the scheduling is static, since after the first divides we must simply map the data on the available nodes. In this case we pay a price: even in presence of balanced algorithms we may have to map pieces of input data of different size. This side effect is inherently present in algorithms like the quick-sort, but not in the merge-sort: in some sense we are making unbalanced a balanced algorithm (this is not always the case, since we can have a number of processing nodes multiple of the division degree of our algorithm).se fa cambiala

In this work we have choose to implement the run time support for divide and conquer computation according to the map approach, in the next sections we will provide a cost model and the implementation details.

2 Implementative Aspects

In this section we will describe DAClib, a parallel and distribuited programming library for writing divide and conquer-based applications.
This library has been written using ANSI/C++ language. In a layered view it leans over a standard parallel programming library such as MPI.
The abstraction provided by MPI, joined with the intensive use of templates, makes the library a platform-indipendent tool suitable for both parallel and distribuited usages.

2.1 Library structure

The library we implemented is composed of several classes and functions. From a user point of view the API provided by DAClib consists of some template classes.

Chunk a template struct which represents a chunk of data to be elaborated;

Divide, Combine, BaseCase these abstract classes must be extended by the user for providing the algorithm code;

DivideAndConquer this is the fundamental class of the library which executes, with the specified parallelism degree, the divide and conquer algorithm provided by the user.

The type dynamicity is handled via template mechanism, as we will clarify in section 2.4.1.
Because of some problems explained in 1.4 we decided to use a particular ‘algorithm-driven’ map-reduce approach.
With this particular approach we have been able to derive a clear cost model to be associated with the skeleton.

When the DivideAndConquer::start method is invoked the parallel computation begins. For convenience the process with rank 0 is the 'master' of the computation.
As shown in Figure 1, first the process with rank 0 reads the input and executes the initial division phase. Subsequently all processes locally execute the divide and conquer on the respective chunk. At last process 0 collects all partial results and combines them in the final solution, which is written in the output file.

```
if (rank = 0) then : data = readInputFromFile(in);
                     while (|chunks| < |processes|)
                       chunks = divide;
                     send(<chunks, processes>);
else : recv(<chunk, 0>);

partial =<local Divide and Conquer on the chunk>

if (rank = 0) then : while (|results| < |processes|)
                        results = recv(<result>);
                     combine(results);
                     writeOutputIntoFile(out);
else : send(<partial, 0>);
```

Figure 1: Pseudocode of the library behaviour.

2.2 The user perspective

Developing DAClib we gave much importance to the ease of use for the users, so our library provides to the user the abstraction from all non-functional concerns. The user can write its divide and conquer application as if it should run in a sequential way.

As shown in Figure 2 the user must only worry about functional concers, writing the code for dividing, combining and solving the base case.

The parallel computation is executed by calling the start method. Once called, the method performs the reading of the data, splits the computation among the processors, and writes the results into the output file.

The structure of the library makes necessary the usage of MPI tools for the compilation and the deploy of the application.

2.3 Cost model

As said before, the particular structure we utilized for the design of the library, allowed us to derive a clear cost model to be associated with the skeleton.
The cost model is derived from the map-reduce one, with some peculiarities due to the fact that the scatter is 'algorithm driven'.

```
#include "DivideAndConquer.h"

using namespace daclib;
...
int main(int argc, char **argv) \{
        DivideAndConquer<...> dac(2);
        ...
        dac.setDivider(&divide);
        dac.setCombiner(&combine);
        dac.setBaseHandler(&base);

        dac.setInputFile("...");
        dac.setOutputFile("...");
        ...
        dac.start(argc, argv);
        ...
\}
```

Figure 2: Sample of usage.

First let's define the notation that will be utilized:
T_D, T_C and T_B are, respctively, the times required for the divide, combine and base solving functions;
S is the input size;
N is the parallelism degree;
d is the division degree.

The sequential completion time is the time needed for the visit of a d-ary tree in wich, in each node is called the divide function first, and the combine function in a second time:

$$T^{(1)} = S \cdot T_D \cdot \log_d S + S \cdot T_B \cdot \log_d S + S \cdot T_C \cdot \log_d S = S \cdot (T_D + T_B + T_C) \cdot \log_d S \quad (1)$$

The parallel completion time is the time needed for visit of a d-ary tree with N leaves which must be added the times of the communication and the visit of a d-ary tree of size $\left(\frac{S}{N}\right)$ (in the balanced case, for unbalanced algorithms the size is $max_1^N |chunks|$:

$$\begin{aligned} T^{(N)} &= S \cdot (T_D + T_C) \cdot \log_d N + \left(\frac{S}{N}\right) \cdot (T_D + T_C) \cdot \log_d \left(\frac{S}{N}\right) + N \cdot T_{send} \left(\frac{S}{N}\right) = \\ &= S \cdot (T_D + T_C) \cdot \log_d N + \left(\frac{S}{N}\right) \cdot (T_D + T_C) \cdot \log_d \left(\frac{S}{N}\right) + N \cdot T_{setup} + S \cdot T_{trasm} = \\ &= S \cdot (T_D + T_C) \cdot \left(\log_d N + \frac{1}{N} \cdot \log_d \left(\frac{S}{N}\right)\right) + N \cdot T_{setup} + S \cdot T_{trasm} \quad (2) \end{aligned}$$

The optimal paralellism degree can be obtained by making the first derivative wrt N be equal to zero:

$$\bar{N} = \frac{\delta T^{(N)}}{\delta N} = -\frac{S \cdot (T_D + T_C) \cdot \ln S}{N^2 \cdot \ln d} + T_{setup} = -S \cdot (T_D + T_C) \cdot \ln S + T_{setup} \cdot \ln d \cdot N^2 = 0$$

$$\bar{N} = \sqrt{\frac{S \cdot (T_D + T_C) \cdot \ln S}{T_{setup} \cdot \ln d}} \qquad (3)$$

Upper bounds for the efficiency and scalability can be calculated once known communication times:

$$\varepsilon = \frac{\frac{T^{(1)}}{N}}{T^{(N)}} = \frac{\frac{S}{N} \cdot (T_D + T_C) \cdot \log_d S}{\frac{S}{N} \cdot (T_D + T_C) \cdot \log_d S + N \cdot T_{send}\left(\frac{S}{N}\right)} \leq 1 \qquad (4)$$

$$s = \frac{T^{(1)}}{T^{(N)}} = \frac{S \cdot (T_D + T_C) \cdot \log_d S}{S \cdot (T_D + T_C) \cdot \log_d S + N \cdot T_{send}\left(\frac{S}{N}\right)} \leq N \qquad (5)$$

2.4 Implementation issues

Developing DAClib, we have faced with several issues. In the following sections will be described some of the problems we have faced to, and the relative solutions.

2.4.1 Types handling

Considered the typical structure of a divide and combine computation:

$$divide : X \rightarrow (X, X)$$

$$base\ solve : X \rightarrow Y$$

$$divide : (Y, Y) \rightarrow Y$$

the user needs a way for specify functions with types that are known only at compilation time.
For solving this problem we decided to use the C++ templates mechanism.

MPI, for making possible the conversion of data types among different architectures, defines a mapping on languages types and provides a mechanism for declaring new data types. This is very easy to use if the types are known when the program is written.
Unfortunately, using templates, the type which is to be sended is known only at compile time.

This problem is solved in the file *Types.h* in which, through template metaprogramming, we perform a mapping of C++ types onto MPI base data types.

As shown in Figure 3 the mapping is performed through a template sctructure, specialized for all C++ types, with a method returning the corresponding

```
/****************************************
* Structure used for mapping base types on MPI::Datatype
* For user-defined classes/structs returns MPI::BYTE
****************************************/
template <class T>
struct MpiType {
  inline static const MPI::Datatype &get() { return MPI::BYTE; }
};

/****************************************
* Mapping of base types on MPI::Datatype struct
****************************************/
template<>
struct MpiType<char> {
  inline static const MPI::Datatype &get() { return MPI::CHAR; }
};
...
```

Figure 3: Mapping of C++ types onto MPI types through template specialization.

MPI::Datatype.
User-defined classes are mapped onto the MPI::BYTE datatype. These classes will need a particular treatment, which is clearly explained in section 2.4.2.

2.4.2 Serialization

As said before, user-defined classes need a special treatment. In fact these classes cannot be mapped onto MPI base data types. MPI provides a mechanism for committing derived data types, but it is suitable only when the structure of the data type is known. This assumption is not true for user-defined classes.

```
class SerializableData {
public:
  SerializableData() {}
  virtual ~SerializableData() {}

  /* returns the size of the serialized class, expressed in bytes
     */
  virtual int getSize() = 0;

  /* buffer is already allocated into serialize method
   * bufSize returns the size of buffer expressed in bytes*/
  virtual bool serialize(void* buffer, int bufSize) = 0;

  /* buffer is already allocated into deSerialize method
   * bufSize represents the size of buffer expressed in bytes*/
  virtual bool deSerialize(void* buffer, int bufSize) = 0;
};
```

Figure 4:

DAClib delegates the problem of the serialization to the user, providing an abstract class to be extended, shown in Figure 4.

```
template <class T>
struct Chunk
{
  unsigned size; //expressed in number of elements of the buffer
        T* buf;

  Chunk() {
    size = 0;
    buf = NULL;
  }

  virtual ~Chunk() {
    size = 0;
  }
};
```

Figure 5: The definition of a chunk.

Serialized data is packed through MPI_PACK routine and sent as MPI::PACKED type.

2.4.3 MPI inside DAClib

Processes executing a parallel divide and conquer computation communicates chunks of data. A chunk is a structure, defined in *UserClasses.h*, representing a piece of data to be computed, or a partial result. The definition is shown in Figure 5.

Sending such a structure through MPI is not a trivial issue. Since the chunk is a structure with variable size, depending on the size of the data to be sended, a new MPI custom data type should be defined for any new chunk size both on sender and receiver side.
Another possible solution should be splitting the sending of the chunk: send the size first, and subsequently send the data (which size is now known from borh sender and receiver). This solution causes performance degradation, as the communication setup time is paid twice for each communication.
We solved the problem by serializing the chunk through MPI_PACK routine, as shown in Figure 6.

During the testing phase of this work we faced some problem relative to the MPI implementation.
In a first version we used MPI_Send routine for communication between processes: this call is a wrapper which implicitly choses one between MPI_Ssend, MPI_Bsend and MPI_Isend routines for performing the communications. This call generates error if it is used with large amounts of data (the amount depends on the implementation used) because it tries to use the buffered call with a too small buffer.
Afterwards, in order to avoid deadlocks, we passed to MPI_Isend routine, in order to avoid deadlocks. Using this call we noticed a drastic performace decrement. After long inspections we noticed that MPI expected to finish the current calculation to make the pending sends.

```
template <class T>
void send(int destId, Task<T> &task, const MPI::Intracomm& comm,
    int tag, Bool2Type<false>, bool test)
{
  ...
  MPI::INT.Pack(&task.task_id, 1, buf, length, pos, comm); //
      support data
  MPI::INT.Pack(&task.pos, 1, buf, length, pos, comm);
  MPI::INT.Pack(&task.data.size, 1, buf, length, pos, comm); //
      chunk data
  (MpiType<T>().get()).Pack(task.data.buf, task.data.size, buf,
      length, pos, comm);
  ...
  comm.Ssend(buf, pos, MPI::PACKED, destId, tag);
}
```

Figure 6: Serialization and send of a chunk.

Finally we used the synchronous version of the send, which is guaranteed to return when the send is completed.

2.5 Future works

Some possible future improvements, to be implemented, could be:

- some optimizations on communications and initial phase can be performed, such as performing the initial divide and the final combine in parallel as a tree.
- implement load balancing, maybe exploiting data-parallel techniques. In fact, the algorithm driven distribution of the data among the processes, makes the load balancing strictly dependent to the algorithm.
- provide multi-thread support in order to exploit multicores in a more efficient way.
- provide tools for deploy, so that the user can compile and run his own applications abstracting the MPI layer tools.

3 Benchmarks and Test Results

In order to test all the functional aspects and to obtain some quantitative performance results, we have implemented a simple divide and conquer algorithm with coarse grain. It is a merge-sort with a division degree of two and a base case modified so to have a coarse grain computation. The full code of this algorithm can be found on the precedent sections, where all the source code is listed.

The parameter involved in the performance estimation are the *completion time*, the *scalability* and the *efficiency*. Assuming that n is the parallelism

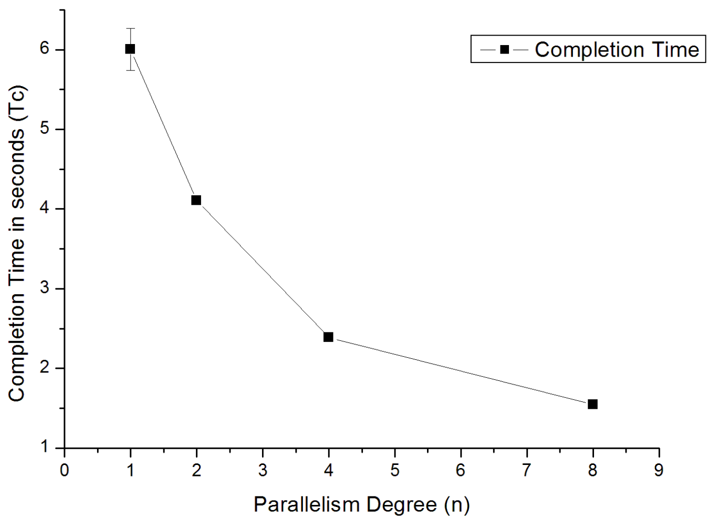

Figure 7: 4M Vector - Completion Time

degree, they are measured as

$$\text{scalability} = \frac{T(1)}{T(n)} \tag{6}$$

$$\text{efficiency} = \frac{T(1)}{nT(n)} \tag{7}$$

Will be now shown the test results obtained running the algorithm on a cluster of dual core Intel machines over a vector of four, eight, forty and eighty megabytes.

As we can see in Figure19 , increasing the problem size lead to good improvement in efficiency and in scalability, especially when the parallelism degree is near the optimal one.
Our library is particulary suitable for solving problems of considerable size with medium to coarse computational grain.

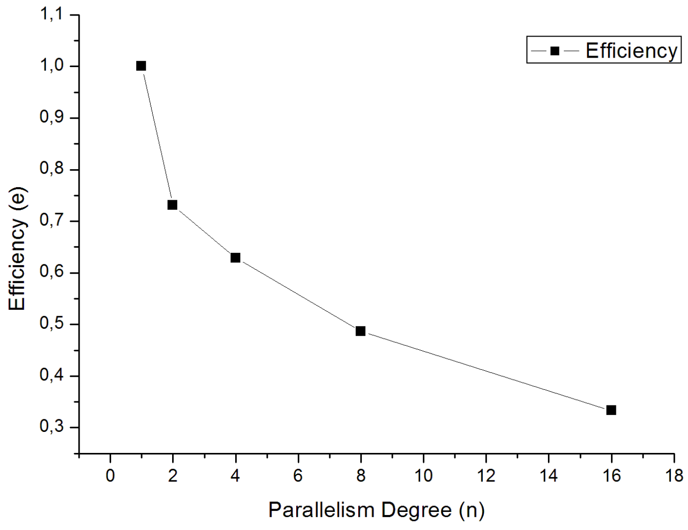

Figure 8: 4M Vector - Efficiency

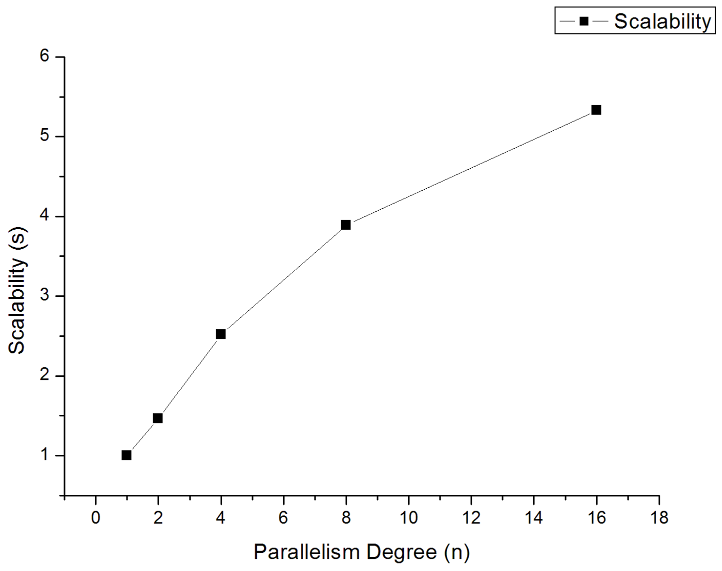

Figure 9: 4M Vector - Scalability

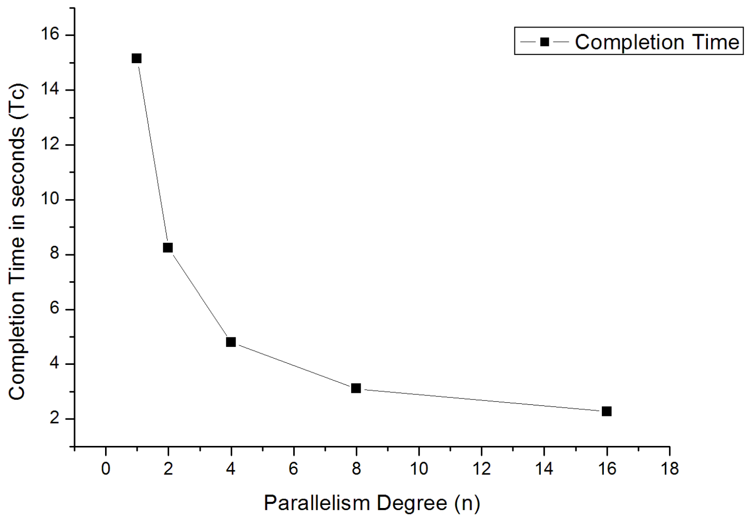

Figure 10: 8M Vector - Completion Time

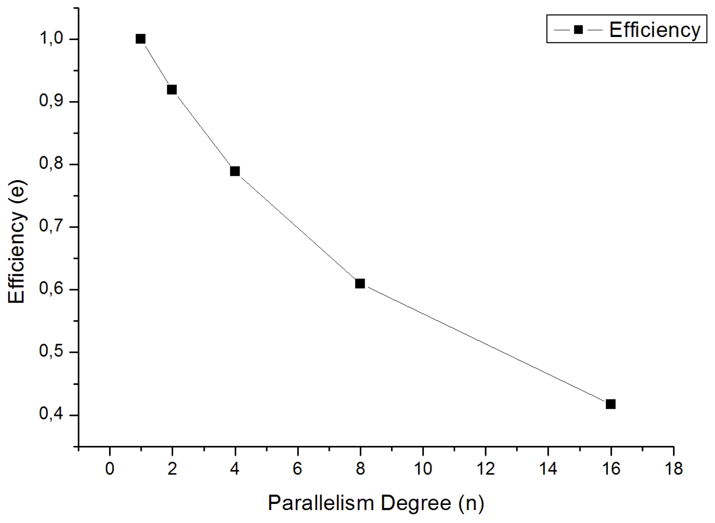

Figure 11: 8M Vector - Efficiency

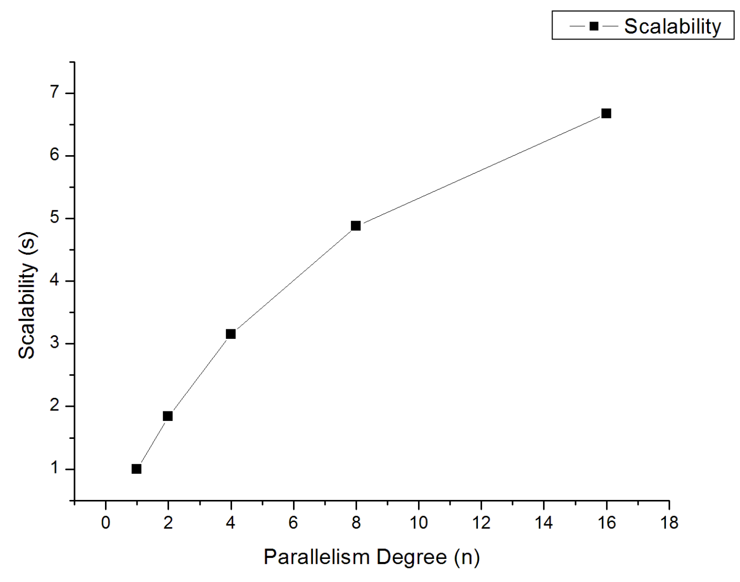

Figure 12: 8M Vector - Scalability

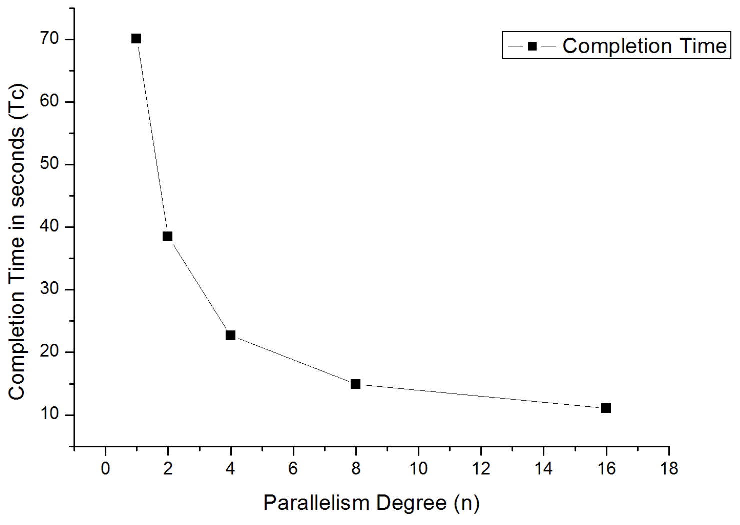

Figure 13: 40M Vector - Completion Time

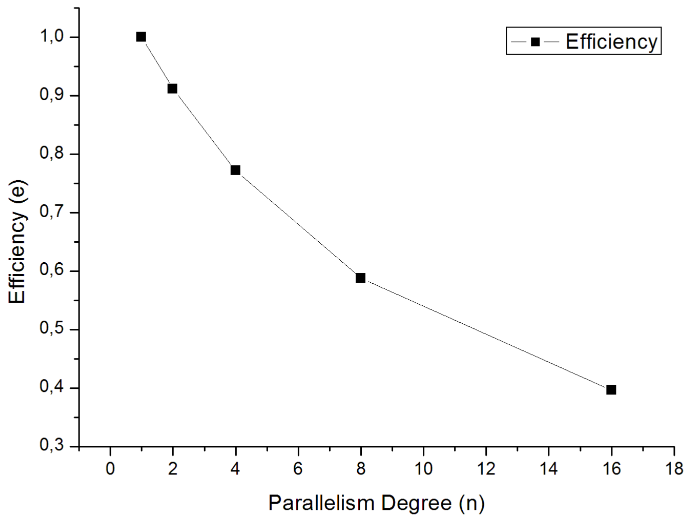

Figure 14: 40M Vector - Efficiency

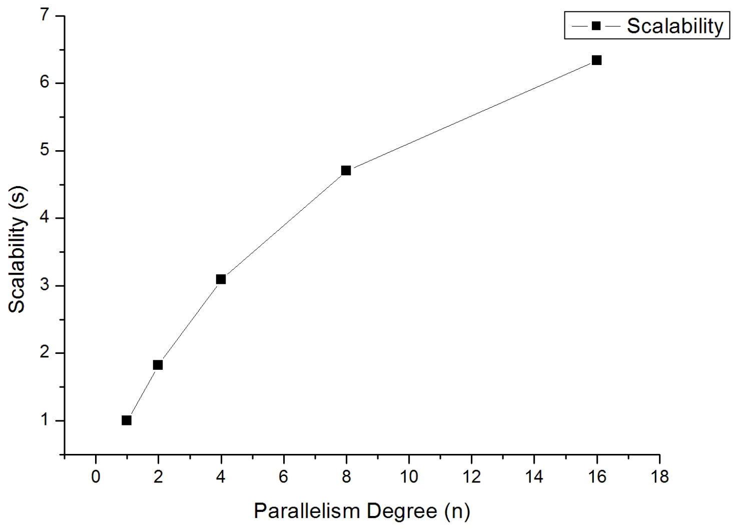

Figure 15: 40M Vector - Scalability

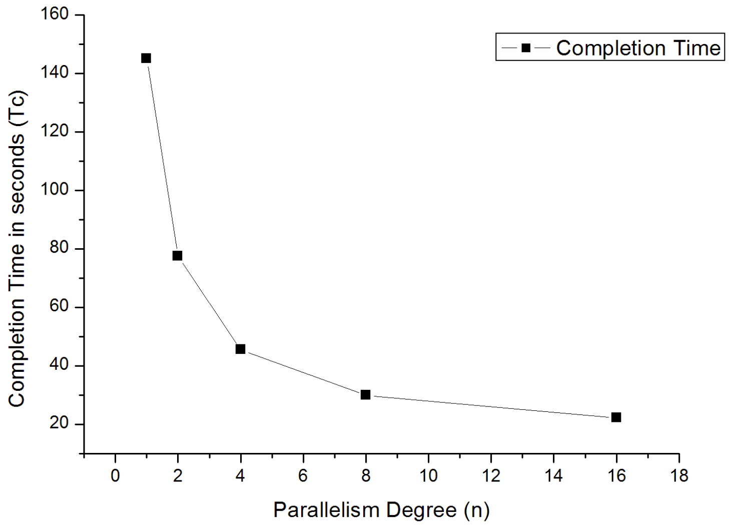

Figure 16: 80M Vector - Completion Time

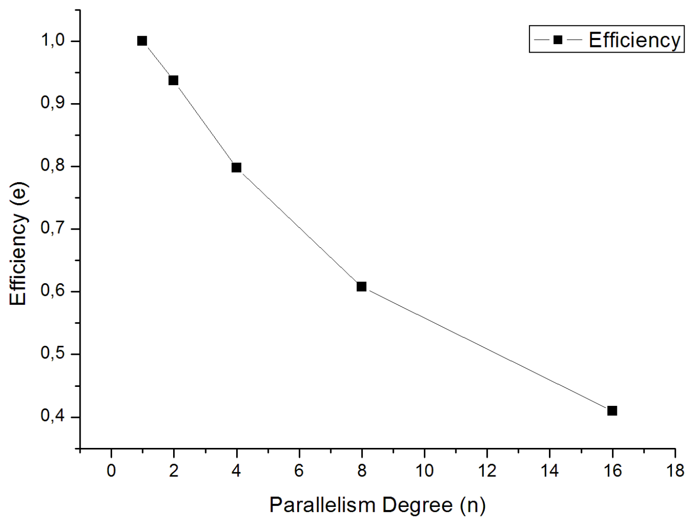

Figure 17: 80M Vector - Efficiency

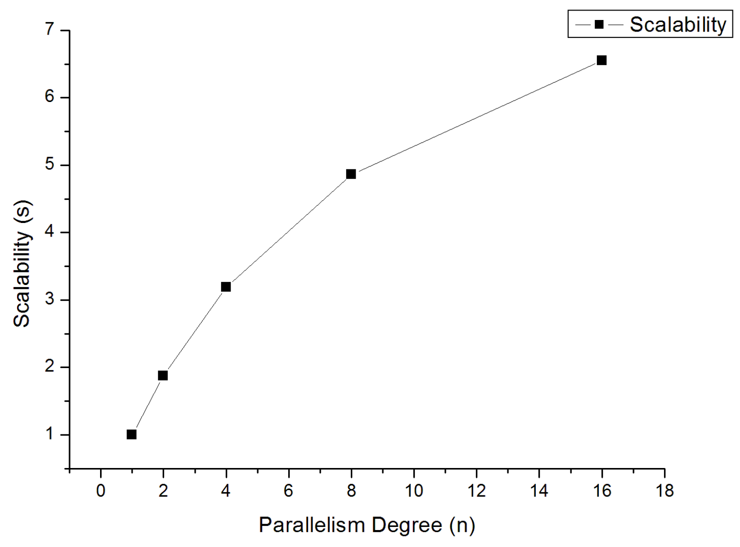

Figure 18: 80M Vector - Scalability

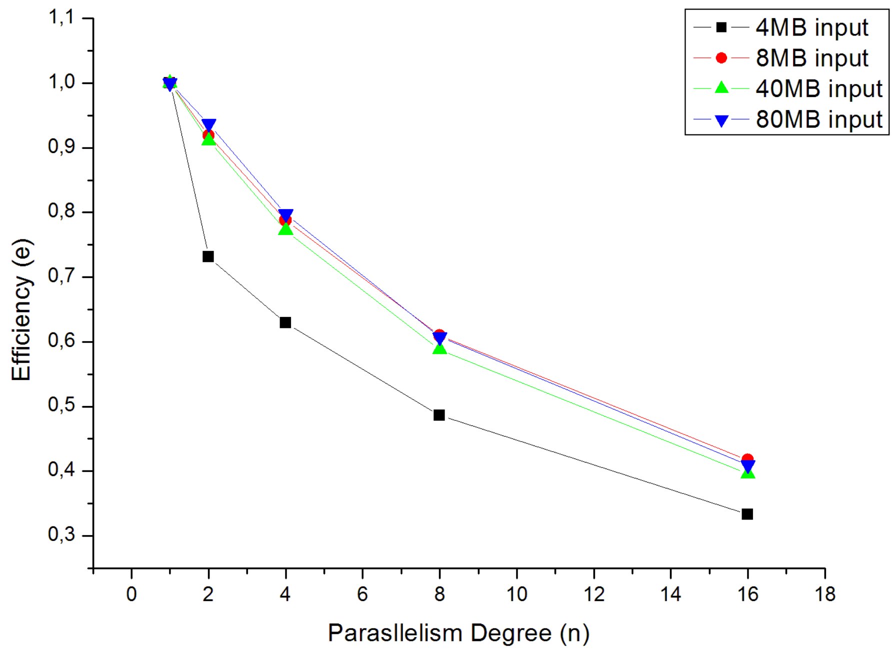

Figure 19: Efficiency in function of the problem size.