Classification of Partioning Problems for Networks of Heterogeneous Computers

Alexey Lastovetsky and Ravi Reddy

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland
{alexey.lastovetsky, manumachu.reddy}@ucd.ie

Abstract. The paper presents a classification of mathematical problems encountered during partitioning of data when designing parallel algorithms on networks of heterogeneous computers. We specify problems with known efficient solutions and open problems. Based on this classification, we suggest an API for partitioning mathematical objects commonly used in scientific and engineering domains for solving problems on networks of heterogeneous computers. These interfaces allow the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition their mathematical objects. These partitioning interfaces are designed to be used along with various programming tools for parallel and distributed computing on heterogeneous networks.

1 Introduction

Parallel solution of regular and irregular problems on a heterogeneous network of computers typically consists of two macro-steps:

- Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes;
- The mapping of these parallel processes to the computers of the network.

An *irregular* problem is characterized by some inherent coarse-grained or large-grained structure. This structure implies a quite deterministic decomposition of the whole problem into relatively small number of subtasks, which are of different size and can be solved in parallel. Correspondingly, a natural way of decomposition of the whole program, which solves the irregular problem on a network of computers, is a set of parallel processes, each solving its subtask and all together interacting via message passing. As sizes of these subtasks are typically different, the processes perform different volumes of computation. Therefore, the mapping of these processes to the computers of the executing HNOC should be performed very carefully to ensure the best execution time of the program.
The most natural decomposition of a *regular* problem is a large number of small identical subtasks that can be solved in parallel. As those subtasks are identical, they are all of the same size. Multiplication of two $n \times n$ dense matrices is an example of a regular problem. This problem is naturally decomposed into n^2 identical subtasks, each of which is to compute one element of the resulting matrix. The main idea behind an efficient solution to a regular problem on a heterogeneous network of computers is to

transform the problem into an irregular problem, the structure of which is determined by the structure of the executing network rather than the structure of the problem itself. So, the whole regular problem is decomposed into a set of relatively large sub-problems, each made of a number of small identical subtasks stuck together. The size of each subproblem, that is, the number of elementary identical subtasks constituting the subproblem, depends on the speed of the processor, on which the subproblem will be solved. Correspondingly, the parallel program, which solves the problem on the heterogeneous network of computers, is a set of parallel processes, each solving one subproblem on a separate physical processor and all together interacting via message passing. The volume of computations performed by each of these processes should be proportional to its speed.
Thus, while the step of problem decomposition is trivial for irregular problems, it becomes key for a regular problem. In fact, at this very step the application programmer designs a heterogeneous data parallel algorithm by working out a generic decomposition of the regular problem parameterized by the number and speed of processors. Most typically the generic decomposition takes the form of data partitioning.
Existing programming systems for heterogeneous parallel computing [1]- [4] support the mapping of parallel algorithms to the executing network but provide very poor support for generic heterogeneous decomposition of regular problems implied by the number and speed of processors. The application programmers need to solve corresponding data partitioning problems and design and implement all supportive code from scratch. Our own experience with using mpC and HMPI for parallel solution regular problems on networks of computers has shown how tedious and error-prone this step of application development can be.
This motivated us to try and automate the step of heterogeneous decomposition of regular problems by designing a library of functions solving typical partitioning problems for networks of heterogeneous computers. Our original approach was to do it by just collecting existing algorithms, designing an API to these algorithms and implementing the API. The main problem we came across on this way was that no classification of partitioning problems was found that might be used as a basis of API design. Existing algorithms created a very fragmented picture. Therefore the main goal of our research became to classify partitioning problems for networks of heterogeneous computers. Such classification had to help to specify problems with known efficient solutions and identify open problems. Then based on this classification an API would have to be designed and partially implemented (for problems that have known efficient solutions). An additional requirement to this classification was that it had to be useful for distributed computing on networks as well.
Our approach to classification of partitioning problems is based on two corner stones:

- A realistic performance model of networks of heterogeneous computers,
- A natural classification of mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) solving problems on networks of heterogeneous computers.

This paper is structured as follows. In section 2, we describe the realistic performance model of networks of heterogeneous computers. In section 3, we identify the mathematical objects. In section 4, we classify the problems encountered during partitioning of sets. Based on this classification, we suggest an API for partitioning sets. Due to limitations on the length of the paper, we only briefly outline the classification of partitioning problems for matrices, graphs, and trees, and the corresponding API.

2 Performance Model of Networks of Heterogeneous Computers

Most algorithms of data partitioning for networks of heterogeneous computers use performance models where each processor is represented by a single positive number that characterizes its relative speed. Data partitioning is performed such that the volume of computations executed by the processor be proportional to its speed.

It is a well known fact that the absolute speed of a processor is a decreasing function of data simultaneously stored in the memory of the processor and used by the processor in computations. The memory typically has a hierarchical structure with levels of fixed sizes. Higher levels are substantially faster and smaller than lower ones. Therefore, as more processed data are stored in the memory, the more levels of the memory hierarchy they fill. As a result more data become stored in slow memory. This increases the average execution time of a single arithmetic operation decreasing the speed of the processor. Figure 1(a) illustrates this fact using matrix multiplication on two computers: modern Dell computer csserver (Linux, main memory of 513960 KB, cache of 512 KB), and relatively old Sun computer csultra01 (Solaris, main memory of 524888 KB, cache of 1024 KB).

Nonetheless the above simple performance model is suitable in many real life situations where the relative speeds of the processors involved in the execution of the application are a constant function of the size of the problem and thus can be approximated by single numbers. Figure 1(b) gives an example of such a situation. The relative speed of computers csserver and csultra01 demonstrated on matrix multiplication may be approximated by a single number, 3, with sufficient accuracy. However if the processors have significantly different sizes at each level of their memory hierarchies, they may demonstrate significantly different relative speeds dependent on the size of the problem. Figure 2 gives us relative speeds of different pairs of computers experimentally obtained during multiplication of matrices of different sizes. If we use such networks

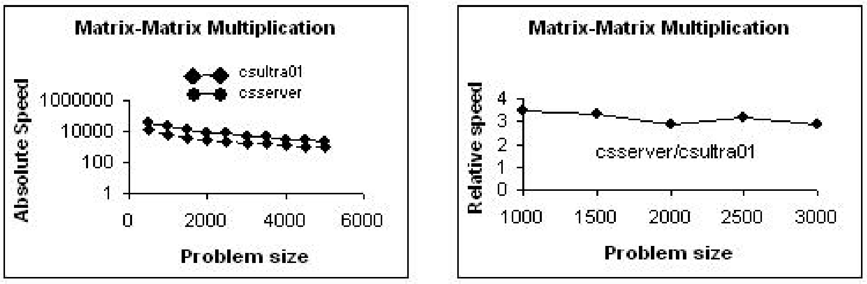

Fig. 1. (a)Absolute Speeds of **csserver** and **csserver01** against the size of the problem in matrix manipulation. (b) The relative speed of these computers against the size of these problems.

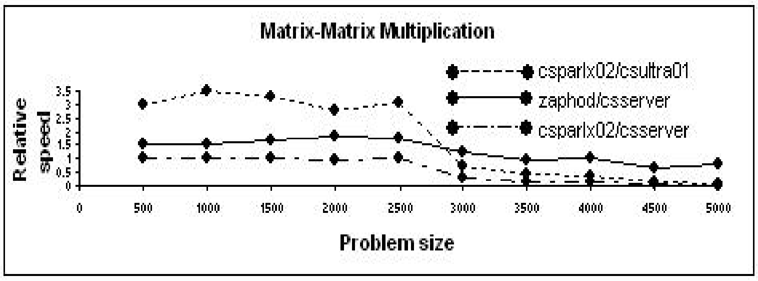

Fig. 2. Relative speeds of computers against the size of the problem in matrix multiplication. Computers involved are: **zaphod**(main memory of 254576 KB, cache of 512 KB), **csparlx02**(126176 KB, 512 KB), **csserver**(513960 KB, 512 KB), **csultra01**(524288 KB, 1024 KB).

of heterogeneous computers for execution of parallel or distributed algorithms, we cannot represent their relative speeds by single numbers. Realistically in this case we must represent the speed by a function of the size of the problem.

Therefore, we suggest using a more realistic model that takes into account the impact of heterogeneity of memory and memory hierarchies on performance.Under this model, each processor is represented by a decreasing function of the problem size that characterizes its speed. In practice, the function is obtained by interpolation of a relatively small number of experimental results for different problem sizes. Constant functions will be just a special case. In addition, the model takes account of memory limitations and characterizes each processor by the maximal size of problem it can solve. The latter feature makes little sense when computing on a local network because in this case the user has some idea about the power of available computers and the size of problem that can be solved on the network. This feature does make sense when the user solves problems on a global network. In that case, the user may have no idea of the number and configurations of computers that may be involved in computations. Therefore if the problem size is big enough, some computer whose speed is estimated based on a small number of experiments may be assigned to solve a subproblem of the size that cannot be solved on the computer at all.

3 Classification of Partitioning Problems

The core of scientific, engineering or business applications is the processing of some mathematical objects that are used in modeling corresponding real-life problems. In particular, partitioning of such mathematical objects is a core of any data parallel algorithm. Our analysis of various scientific, engineering and business domains resulted in the following short list of mathematical objects commonly used in parallel and distributed algorithms: **sets** (ordered and non-ordered), **matrices** (and multidimensional arrangements), **graphs** and **trees**.

These mathematical structures give us the second dimension for our classification of partitioning problems. In the next section, we present our approach to classification of partitioning problems using sets as mathematical objects. We also suggest an API based on the classification.

4 Partitioning Problems for Sets and Ordered Sets

There are two main criteria used for partitioning a **set**:

a) The number of elements in each partition should be proportional to the speed of the processor owning that partition.
b) The sum of weights of the elements in each partition should be proportional to the speed of the processor owning that partition.

Additional restrictions that may be imposed on partitioning of an **ordered set** are:

- The elements in the set are well ordered and should be distributed into disjoint contiguous chunks of elements.

The most general problem of partitioning a set can be formulated as follows:

- Given: (1) A set of **n** elements with weights $\mathbf{w_i}$ (i=0,…,n-1), and (2) A Well ordered set of **p** processors whose speeds are functions of the size of the problem, $\mathbf{s_i} = \mathbf{f_i}(\mathbf{x})$, with an upper bound $\mathbf{b_i}$ on the number of elements stored by each processor (i=0,…,p-1),
- Partition the set into **p** disjoint partitions such that: (1) The sum of weights in each partition is proportional to the speed of the processor owning that partition, and (2) The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it.

The most general partitioning problem for an ordered set can be formulated as follows:

- Given: (1) A set of **n** elements with weights $\mathbf{w_i}$ (i=0,…,n-1), and (2) A Well ordered set of **p** processors whose speeds are functions of the size of the problem, $\mathbf{s_i} = \mathbf{f_i}(\mathbf{x})$, with an upper bound $\mathbf{b_i}$ on the number of elements stored by each processor (i=0,…,p-1),
- Partition the set into **p** disjoint contiguous chunks such that: (1) The sum of weights of the elements in each partition is proportional to the speed of the processor owning that partition, and (2) The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it.

The most general partitioning problems for a set and an ordered set are very difficult and open for research. At the same time, there are a number of important special cases of these problems with known efficient solutions. The special cases are obtained by applying one or more of the following simplifying assumptions:

- All elements in the set have the same weight. This assumption eliminates **n** additional parameters of the problem.
- The speed of each processor is a constant function of the problem size.
- There are no limits on the maximal number of elements assigned to a processor.

Table 1. Special cases of partioning of a set

Mode of Parallel Computation	**Weights of elements are the same**	**Weights of elements are different**
Speeds are functions of problem size & no limits on number of elements stored by each processor.	**Complexity** O(p$\times\log n$)	No Known Results
Speeds are single constant numbers and an upper on number of elements stored that each processor can hold.	**Complexity** O(p)	NP-Hard?
Speeds are single constant numbers & no limits on number of elements stored that each processor can hold.	**Complexity** O(p)	NP-Hard?

One example of a special partitioning problem for a set is:

- Given: (1) A set of **n** elements, and (2) A well-ordered set of **p** processors whose speeds are represented by single constant numbers, s_0, s_1,..., s_i.
- Partition the set into **p** disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that partition.

This problem is trivial of the complexity O(p). Another example of a special partitioning problem for a set is:

- Given: (1) A set of **n** elements, and (2) A well-ordered set of **p** processors whose speeds are functions of the size of the problem, $\mathbf{s_i} = \mathbf{f_i}(\mathbf{x})$ (i=1,...,p-1).
- Partition the set into **p** disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that partition.

An algorithm of the complexity O (p $\times \log n$) solving this problem is given in [5].
Table 1 and Table 2 summarize specific partitioning problems for a set and an ordered set respectively and their current state to the best knowledge of the authors.
Based on this classification, we suggest the following API to application programmers for partitioning a set into **p** disjoint partitions:

> **typedef double** (*User_defined_metric)(**int** p, **const double** *speeds,
> **const double** *actual, **const double** *ideal);
> **int** Partition_set (**int** p,**int** pn, **const double** *speeds, **const int** *psizes,
> **const int** *mlimits, **int** n, **const double** *w **int** ordering, **int** processor_reordering,
> **int** type_of_metric, User_defined_metric umf, **int** *metric, **int** *np)

Parameter **p** is the number of partitions of the set. Parameters **speeds** and **psizes** specify speeds of processors for **pn** problem sizes. These parameters are 1D arrays of size **p**$\times$**pn** logically representing 2D arrays of shape **[p][pn]**. The speed of the **i**-th processor for **j**-th problem size is given by the **[i][j]**-th element of **speeds** with the problem size itself given by the **[i][j]**-th element of **psizes**. Parameter **mlimits** gives the maximum

Table 2. Special cases of partioning of an ordered set

Mode of Parallel Computation	**Weights of elements are the same**	**Weights of elements are different**	
		Rearrangement of Processors	
		Allowed	**Not allowed**
Speeds are functions of the size of the problem & no upper bound exists on number of elements that each processor can hold.	**Complexity**	No Known Results	No known Results
	O(p$\times \log n$)		
Speeds are single constant numbers & an upper bound exists on number of elements that each processor can hold.	**Complexity**	No Known Results	No known Results
	O(p)		
Speeds are single constant numbers & no limits exist on number of elements stored that each processor can hold.	**Complexity**	No Known Results	No known Results
	O(p)		

number of elements that each processor can hold.
Parameter **n** is the number of elements in the set, and parameter **w** is the weights of its elements. If **w** is **NULL**, then the set is partitioned into **p** disjoint partitions such that criterion (a) is satisfied. If parameters **w, speeds** and **psizes** are all set to **NULL**, then the set is partitioned into **p** disjoint partitions such that the number of elements in each partition is the same. If **w** is not **NULL**, then the set is partitioned into **p** disjoint partitions such that criterion (b) is satisfied. If **w** is not **NULL** and **speeds** is **NULL**, then the set is partitioned into **p** equally weighted disjoint partitions.
Parameter **ordering** specifies if the set is well ordered (=1) or not (=0).
Parameter **type_of_metric** specifies whose metric should be used to determine the quality of the partitioning. If **type_of_metric** is **USER_SPECIFIED**, then the user provides a metric function **umf**, which is used to calculate the quality of the partitioning. Otherwise, the system-defined metric is used which is the weighted Euclidean metric. The output parameter **metric** gives the quality of the partitioning, which is the deviation of the partitioning achieved from the ideal partitioning satisfying the partitioning criteria.
If **w** is **NULL** and the set is not ordered, the output parameter **np** is an array of size **p**, where **np[i]** gives the number of elements assigned to the **i**-th partition. If the set is well ordered, processor **i** gets the contiguous chunk of elements with indexes from **np[i]** upto **np[i]+np[i+1]-1**.
If **w** is not **NULL** and the set is well ordered, then the user needs to specify if the implementations of this operation may reorder the processors before partitioning (Boolean parameter **processor_reordering** is used to do it). One typical reordering is to order the processors in the decreasing order of their speeds.
If **w** is not **NULL**, the set is well ordered and the processors cannot be reordered, then

the output parameter **np** is an array of size **p**, where **np[i]** gives the number of elements of the set assigned to the **i**-th partition. Specifically, processor **i** gets the contiguous chunk of elements with indexes from **np[i]** upto **np[i]+np[i+1]-1**.
If **w** is **NULL**, the set is well ordered and the processors may be reordered, then **np** is an array of size **2**$\times$**p**, where **np[i]** gives index of a processor and **np[i+1]** gives the size of the contiguous chunk assigned to processor given by the index **np[i]**.
If **w** is not **NULL** and the set is not ordered, then **np** is an array of size **n**, containing the partitions to which the elements in the set belong. Specifically, **np[i]** contains the partition number in which element **i** belongs to.
Some of the typical examples where the partitioning interfaces for sets can be used are striped partitioning of a matrix and simple partitioning of a graph. In striped partitioning of a matrix, a matrix is divided into groups of complete rows or complete columns, the number of rows or columns being proportional to speeds of the processors. In simple partitioning of an unweighted graph, the set of vertices are partitioned into disjoint partitions such that the criterion (a) is satisfied. In simple partitioning of a weighted graph, the set of vertices are partitioned into disjoint partitions such that criterion (b) is satisfied.

5 Conclusion

The same approach is applied to classification of partitioning problems for matrices, graphs, and trees. More information on partitioning these mathematical objects and related API can be found in [6].

References

1. Arapov, D., Kalinov, A., Lastovetsky, A., Ledovskih, I.: A Language Approach to High Performance Computing on Heterogeneous Networks. Parallel and Distributed Computing Practices 2(3), pp.87-96, 1999
2. Lastovetsky, A., Arapov, D., Kalinov, A., Ledovskih, I.: A Parallel Language and Its Programming System for Heterogeneous Networks. Concurrency: Practice and Experience 12(13), pp.1317-1343, 2000
3. Lastovetsky, A.: Adaptive Parallel Computing on Heterogeneous Networks with mpC. Parallel Computing 28(10), pp.1369-1407, 2002
4. Lastovetsky, A., Reddy,R.: HMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers. In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), CD-ROM/Abstracts Proceedings, IEEE Computer Society 2003
5. Lastovetsky, A., Reddy, R.: Towards a Realistic Model of Parallel Computation on Networks of Heterogeneous Computers. Technical Report, University College Dublin, April 2003
6. Lastovetsky, A., Reddy, R.: Classification of Partitioning Problems for Networks of Heterogeneous Computers. Technical Report, University College Dublin, December 2003