Isidoro Gitler · Jaime Klapp (Eds.)

High Performance Computer Applications

6th International Conference, ISUM 2015
Mexico City, Mexico, March 9–13, 2015
Revised Selected Papers

Springer

Editors
Isidoro Gitler
CINVESTAV-IPN
ABACUS Centro de Matemáticas Aplicadas y Cómputo de Alto Rendimiento
La Marquesa
Mexico

Jaime Klapp
Instituto Nacional de Investigaciones Nucleares
La Marquesa
Mexico

ISSN 1865-0929 ISSN 1865-0937 (electronic)
Communications in Computer and Information Science
ISBN 978-3-319-32242-1 ISBN 978-3-319-32243-8 (eBook)
DOI 10.1007/978-3-319-32243-8

Library of Congress Control Number: 2016935218

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

Contents

An Agglomeration Strategy for the Parallel Processes Mapping onto a Distributed Computing Architecture

Juan C. Catana-Salazar[1(✉)] and Jorge L. Ortega-Arjona[2]

[1] Posgrado en Ciencia e Ingeniería de la Computación, Universidad Nacional Autónoma de México, Mexico City, Mexico
j.catanas@uxmcc2.iimas.unam.mx

[2] Departamento de Matemáticas, Facultad de Ciencias, Universidad Nacional Autónoma de México, Mexico City, Mexico
jloa@ciencias.unam.mx

Abstract. Parallel processes, by nature, tend to interchange a high amount of data between them to maintain a highly cohesive system. Nevertheless, when a parallel system is executed on a distributed computing architecture, communications over the network and the time spent by them become very important.

This paper introduces a strategy, based on the Max-flow min-cut theorem, to agglomerate and allocate parallel processes onto a distributed computing architecture. The main goal of the strategy is to decrease the amount of remote communications and increase the amount of local communications. The strategy allocates the processes "carefully" over the distributed nodes, and that causes the communication time of the parallel system to be minimized.

Keywords: Parallel process · Mapping problem · Networks flows

1 Introduction

A parallel system is a set of processes that communicate with each other and collaborate to accomplish a common goal. Parallel systems not only have multiple instruction flows executing at the same time, but also multiple data flows between processes [7].

A parallel system can be classified depending on its communication or synchronization needs. The granularity $g(p_i)$ is a qualitative measure obtained by dividing the processing time $t_{proc}(p_i)$ and communication time $t_{com}(p_i)$ of a process p_i [5], see Eq. 1.

$$g(p_i) = \frac{t_{proc}(p_i)}{t_{com}(p_i)} \tag{1}$$

Three types of granularity derived from the relation between processing and communication times are shown next:

© Springer International Publishing Switzerland 2016
I. Gitler and J. Klapp (Eds.): ISUM 2015, CCIS 595, pp. 201–212, 2016.
DOI: 10.1007/978-3-319-32243-8_14

1. **Fine granularity:** Says a process is fine grained if $t_{com} > t_{proc}$.
2. **Medium granularity:** A process is medium grained when $t_{com} \simeq t_{proc}$.
3. **Coarse granularity:** Says a process is coarse grained when $t_{com} < t_{proc}$.

Any parallel machine or multiprocessor system must implement communications via one or more memory blocks. There is a broad variety of memory architectures, which mainly differ on the memory access method [9]. Two widely used memory architectures are the following:

1. **Shared memory:** Memory is directly accessed, commonly through a bus, by every processor in the system. Every processor has a common "snapshot" of the shared memory [7].
2. **Distributed memory:** There are many memory blocks housing many processes. The processes hosted in a memory block can only "see" the local data. The processes, hosted in different memory blocks, needs a network channel to interchange data [7,9].

The main difference between memory architectures is the communication time, in the shared memory architecture communication time is fast and uniform in access time, due to the "closeness" between memory and processors. On the other hand in a distributed memory architecture, communication time is variable and depends on external characteristics related with the network channel, network protocols etc. [7].

The execution time $ET(P)$ of a parallel system P can be defined (in a very simplified way) as, the processing time $PT(P)$ plus the communication time of the system $CT(P)$.

For each communication $c(p_i, p_j)$ between two processes p_i and p_j, such that both are hosted at the same memory block. A constant communication time, t_{cons} is added to the execution time $ET(P)$. Additionally, for each communication $c(p_k, p_l)$ between two processes p_k and p_l, such that both are hosted on different memory blocks. A variable communication time t_{var} is added to the execution time $ET(P)$, where:

$$t_{cons} << t_{var} \tag{2}$$

Let $c_l(P)$ be the amount of local communications and $c_r(P)$ the amount of remote communications of P. Then $CT(P)$ is equal to $c_l(P)$ constant time communications plus $c_r(P)$ variable time communications. See Eq. 3.

$$ET(P) = PT(P) + c_l(P) * t_{cons} + c_r(P) * t_{var} \tag{3}$$

This paper introduces a strategy, based on the Max-flow min-cut theorem, to agglomerate and allocate the processes of a parallel system onto a distributed computing architecture with k processing nodes. The main goal of the strategy is to minimize the communication time $CT(P)$ of P, by increasing $c_l(P)$ the amount of local communications and decreasing $c_r(P)$ the amount of remote communications among processes. The mapping of the parallel processes is determined in a statical off-line manner. The balanced workload on the processing nodes, is not considered by this strategy.

In Sect. 2 a set of definitions are introduced to help establish a few tools needed by the strategy proposed in this work. Section 3 describes the agglomeration problem and presents additional considerations. Section 4 shows the strategy in a detailed way. Section 5 shows a case study where the proposed strategy is applied in a real life case. Finally Sect. 6 shows the conclusions of this work.

2 Background

In this section are presented two main themes, parallel software methodology and network flows.

2.1 Parallel Software Methodology

There are many parallel software methodologies proposed in the literature. The common goal of every methodology is to have an easy way to translate a sequential problem into a parallel system. It is also desirable to consider factors such as performance and efficiency.

The following four steps, can be found in every parallel software methodology:

1. **Partitioning or Decomposition.** The partitioning stage involves the division of a general problem into a set of independent modules that can be executed in parallel. That does not imply having the same number of processes as the number of processors. This stage is concerned with finding parallelism in every opportunity, regardless of the resources available to the system [4,5].
2. **Communication.** The parallel modules, generated by the previous stage, can be executed concurrently but not independently. Every parallel process is linked to data provided by other tasks, so that data is propagated along the parallel system [1]. Local communications implies two geographically close communicating processes. In contrast, remote communication implies two processes that communicate through a network medium [1].
3. **Agglomeration and Granularity Adjustment.** The agglomeration stage is responsible of controlling the granularity, either to increase the parallel processing or decrease the communication costs. The main idea is to use the locality, i.e., to group some tasks in a way that reduces communication over the network [4].
4. **Mapping.** The parallel processes must be mapped or allocated into a set of processors to be executed, this is called the mapping problem. Also it is defined as the problem of maximize the number of communicating processes pairs allocated in two directly connected processors [2,4]. The allocation can be specified statically or can be determined in execution time [3].

2.2 Network Flows

A network flow $G_{nf} = (V, E)$ is a strictly directed graph, where each edge $a = (u, v) \in E$ has a capacity $c(a) \geq 0$. If E has an edge $a = (u, v)$, then there is not an edge $a_r = (v, u)$ in the opposite direction [6].

A network flow has two special vertices, a source vertex s, and a target vertex t. The source vertex is responsible of generating the flow to be routed through the edges of the network [3,6].

Assume that, for all $v \in V$ there is a *path* $s \to v \to t$, i.e. the graph is connected. Note that $v \in V - \{s\}$ has at least one incident edge, then $|E| \geq |V| - 1$ [6].

There are two interesting problems in this type of graphs, which are presented below:

1. **Min Cut Problem:** An $s - t$ cut, where $s \in A$ and $t \in B$, is a partition of the set V into two groups $V = \{A, B\}$.

   The capacity of a cut is defined as $c(A, B)$, which is equal to the sum of capacities of each edge $e \in E$ that goes out from A [3].

$$cap(A, B) = \sum_{e \text{ goes out } A} c(e) \tag{4}$$

   The minimum cut problem refers to find an $s-t$ cut of minimum capacity [3].
2. **Max Flow Problem:** An $s - t$ flow is defined as a function that satisfies two properties [3]:
   (a) **Capacity:** All flow assigned to an edge e should be less or equal than its capacity $c(e)$.

$$0 \leq f(e) \leq c(e), \ \ \forall\, e \in E \tag{5}$$

   (b) **Preservation:** The total flow entering to a vertex $v \in V - \{s, t\}$, should be equal to the total flow coming out from it.

$$\sum_{e \text{ goes to } v} f(e) = \sum_{e' \text{ goes out } v} f(e'), \ \ \forall\, v \in V(G) - \{s, t\} \tag{6}$$

   The *maximum flow problem* refers to finding an $s-t$ flow of maximum value, with no infringement of the capacity and preservation properties [3,6].

Every edge $e = (u, v)$ in a network flow G_{nf} has a residual edge $e_r = (v, u)$ associated to it, such that $c(e_r) = f(e)$. The residual edge is allowed to transfer flow directed to the target vertex t, so that when a flow g pass through a residual edge e_r then $f(e) = f(e) - g$.

It turns out that the *min cut problem* and the *max flow problem* are closely related, and it is shown by the next lemma.

The *net flow* across a cut (A, B), is the sum of the flow on the edges that goes from A to B, minus the sum of the flow on the edges that goes from B to A.

Lemma 1 (Flow Value). *Let f be any flow assigned to edges of E. Let (A, B) any $s - t$ cut from the network flow. Then, the net flow sent across the cut is equal to the value of f [3].*

$$val(f) = net(f) = \sum_{e \text{ goes out } A} f_{out}(e) - \sum_{e \text{ enters to } A} f_{in}(e) \quad (7)$$

By the previous lemma is easy to see the duality of this two problems, such that the maximum value of a flow f, across a $s-t$ cut, should be less or equal than the minimum cut's capacity on the network.

$$val(f) = net(f) \; across \; (A, B) \leq c(A, B) \quad (8)$$

There exists an algorithm to find the max flow and the min cut over a network flow called Ford-Fulkerson algorithm. The algorithm is based in one important concept called *augmenting path*. An augmenting path is a simple directed path, from s to t, with positive capacities edges, such that, the flow over the network can be increased [6].

Theorem 1 Augmenting Path Theorem. *A flow f, which is obtained by the Ford-Fulkerson algorithm, is maximum if, there is no more augmenting paths in the network flow [6].*

As a corollary by previous theorem and lemmas:

Theorem 2 Maximum Flow Minimum Cut Theorem. *Let f be a $s-t$ flow such that there is no an augmented path in the graph G. Let (A, B) be an $s-t$ cut in G such that $net(f) = c(A, B)$. f is the maximum flow value in G, and $c(A, B)$ is the minimum capacity for every $s-t$ cut in G [6].*

3 The Agglomeration Problem

As mentioned in the *Agglomeration and Granularity Adjustment* stage of the methodology presented in Sect. 2.1, is in this step where the problem of minimizing the communication costs of a parallel system can be addressed.

In order to agglomerate a set of parallel processes, it is necessary to group some of them in accordance to a given criterion, for the purposes of this work the main criterion is the minimization of communication costs.

The agglomeration problem can be seen as a more general problem called graph partitioning. Most partitioning problems are known to be *NP-Hard*, meaning that there is no efficient way to solve them. Instead, the use of heuristics and approximation algorithms has been proposed as a solution [12].

The partitioning problem is defined as follows: Let $G = (V, E)$ be a graph with weighted edges, where $|V| = n$. The (k, v)-balanced partitioning problem, for some $k \geq 2$, aims to decompose G into subsets at most size $v\frac{n}{k}$. Where the aggregated weight of the k edges connecting two vertices from different components is minimal [12].

In particular, the k-balanced partitioning problem is shown as a *NP-Complete* problem in [12]. Even the $(2, 1)$-balanced partitioning problem, which seems to be more easy, is also an *NP-Complete* problem [8,10].

There are two main approaches of the approximation algorithms for graph partitioning. The local algorithms, which make decisions based on local search strategies, such as the Kernighan-Lin algorithm [13] and the Fiduccia-Mattheyses algorithm [14], and the global algorithms that rely on properties of the entire graph, the best known is the spectral partitioning algorithm [15].

The agglomeration problem, as here is defined with no load balance considerations, can be directly addressed by the minimum k-cut problem. The minimum k-cut problem asks for a minimum set of weighted edges whose removal leaves k connected components [11].

The minimum set of weighted edges, to disconnect a graph into two components, can be found in polynomial time by the Ford-Fulkerson algorithm [3]. For a k decomposition of the graph, compute the minimum cut of each subgraph and take the lightest one, repeat until there are k connected components. The previous algorithm guarantees a $2 - \frac{2}{k}$ approximation [11].

3.1 The Minimum Communication Cut Algorithm

Consider the network flow shown in Fig. 1. Such graph has a minimum cut shaped by edges $(s, 2)$ and $(3, 5)$ of value 19.

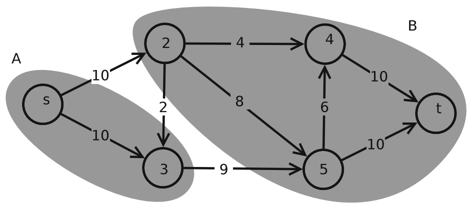

Fig. 1. Network Flow. Grey shapes represent the two partitions (A, B) computed by the Ford-Fulkerson algorithm.

Note that edge $(2, 3)$ is incident to the partition A, and its weight is not considered for the value of the flow.

Remark 1. By definition, the capacity of a cut does not consider any edge incident to the partition A, but in the context of parallel processes such edges are communications among processes of the system, such that, the flow transmitted across those edges must be considered by the agglomeration step.

Therefore it can be said that:

Definition 1. *A minimum communication cut, for the parallel processes agglomeration problem, is the sum of capacities of edges directed to the partition A plus the sum of capacities of edges directed to the partition B.*

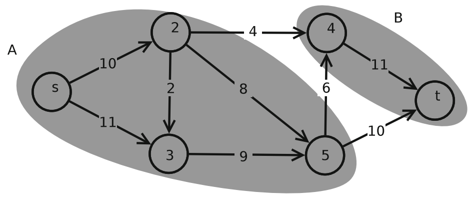

Fig. 2. Minimum communication cut from the network flow shown in Fig. 1.

Taking Remark 1 into account, the communication cut shown in Fig. 1 is of value 21. And the minimum communication cut, of the same graph, is given by the edges $(2, 4)$, $(5, 4)$ y $(5, t)$ of value 20 (see Fig. 2).

Algorithm 1, takes as input parameters a network flow G, a source vertex s, and a target vertex t. It gives as a result the minimum communication cut of the graph G.

The Algorithm 1 first compute the minimum cut of G by using the Ford-Fulkerson algorithm. Then is replaced every edge e in $mincut(G)$, such that e is incident to the partition A, by its inverse edge. Finally, is repeated the computation of the minimum cut over the new G until get a minimum cut with no incident edges to the partition A.

Algorithm 1. Minimum Communication Cut(G, s, t)

```
$mincut(G) \leftarrow$ Ford-Fulkerson($G, s, t$)
**while** $\exists\, e \in mincut(G)$ incident to $A$ **do**
  **for all** $e \in mincut(G)$ incident to $A$ **do**
    $G \leftarrow G - e$
    $G \leftarrow G + e_{inverse}$
  **end for**
  $mincut(G) \leftarrow$ Ford-Fulkerson($G, s, t$)
**end while**
```

Note that reversing an edge whose flow is greater than zero may cause the violation of conservation and capacity properties. Because of that, is important to prove the next lemma.

Let $G = (V, E)$ be a network flow, and $mincut(G) = \{e_1, e_2, ..., e_k\}$ the set of edges in the minimum cut of G.

Lemma 2. *For all edge $e = (y, x) \in mincut(G)$, where $y \in B$ y $x \in A$, has an assigned flow $f(e) = 0$.*

Proof. By contradiction suppose that $f(e) > 0$.

Note that for all edge $e' \in mincut(G)$, that goes from the partition A to the partition B, has a flow $f(e') = c(e')$, otherwise it would not be a minimum cut.

W. l. g. suppose there is at least one edge e that goes from the partition B to partition the A, which by assumption has a flow $f(e) > 0$. Therefor, there is a residual edge e_r assigned to e with $c(e_r) > 0$, there are two cases:

1. There is at least one path P_{s-t} that uses e_r to transfer flow from the vertex s to the vertex t, thus $mincut(G)$ is not a minimum cut of G.
2. There is no path P_{s-t} that uses e_r to transfer flow from the vertex s to vertex t, meaning that there is one $mincut(G)'$ such that $c(mincut(G)') < c(mincut(G))$.

Any case contradicts the assumptions $\Rightarrow\Leftarrow$. $\square$

4 The Agglomeration Strategy

In this section the agglomeration strategy for parallel processes mapping is introduced.

4.1 Building the Software Graph

Consider the parallel processes defined in the *decomposition* stage of the methodology presented in Sect. 2.1, such processes can be represented as a set of vertices V_{sw}.

Let E_{sw} be the set of edges that represent the relations between two processes established in the *communication* stage, such that, for all $p_i, p_j \in V_{sw}$ there is an edge $e_{p_i-p_j}$ if this processes interchange $d > 0$ units of data. For all $e_{p_i-p_j} \in E_{sw}$ there is an associated capacity $c(e_{p_i-p_j}) = d$

An edge capacity is equal to the sum of data units interchanged by two processes during execution time. The number of data units is totally dependent on the nature of the parallel system. The amount of communications must be well defined and be representable as a positive integer, otherwise there is no sense in agglomerating by using this strategy.

Let $G_{sw} = \{V_{sw}, E_{sw}\}$ be the software graph that represents a parallel system P.

4.2 Transformation to a Network Flow

In order to transform the software graph into a network flow, is necessary to classify the vertices of G_{sw} considering the following criterion:

- **Initial Vertices or Flow Generators:** Most of them are main processes which generate and send data to other processes in the system. Sometimes they communicate with each other or do some processing.
- **Dealers or Flow Distributors:** They receive data from flow generator vertices. Usually they do some processing but their main function is to distribute data across end vertices.

– **End Vertices or Processing Vertices:** They represent slave processes, and their main purpose is to do processing tasks with data received from initial or dealer vertices. Sometimes they communicate with each other.

Let $G_{nf} = \{V_{nf}, E_{nf}\}$ be a network flow, such that G_{nf} is directed and has no parallel edges. Then:

1. Let s, t be two vertices, such that s is a source vertex and t is target vertex, then:
$$V_{nf} = V_{sw} \cup \{s, t\} \tag{9}$$
2. The vertex s should have a directed edge e_g for every generator vertex v_{gen} in the software graph, such that:
$$E_{nf} = E_{sw} \cup \{e_g\} \mid e_g = (s, v_{gen}) \;\forall\, v_{gen} \in V_{sw} \tag{10}$$
The capacity $c(e_g)$ of every edge $e_g = (s, v_{gen})$ should be equal or greater than the sum of capacities of every edge that goes out from v_{gen}.
$$c(e_g) \geq \sum c(e_{out}) \mid e_{out} = (v_{gen}, u), \;\forall\, e_{out} \in v_{gen} \tag{11}$$
3. For every end vertex v_f, there is a directed edge e_f ending at t:
$$E_{nf} = E_{sw} \cup \{e_f\} \mid e_f = (v_{end}, t), \;\forall\, v_{end} \in V_{sw} \tag{12}$$
The capacity $c(e_f)$ of every edge $e_f = (v_{fin}, t)$ should be equal o greater than the sum of capacities of every edge that ends at v_{end}.
$$c(e_f) \geq \sum c(e_{in}) \mid e_{in} = (u, v_{gen}), \;\forall\, e_{in} \in v_{end}. \tag{13}$$

4.3 Applying the Algorithm

Given a network flow G_{nf}, the agglomeration problem can be addressed by using the greedy algorithm to solve the minimum k-cut problem, as described in Sect. 3, and using the Algorithm 1 instead of the traditional Ford-Fulkerson algorithm.

As a result a k agglomeration of the parallel system P, which can be directly allocated into the k processing nodes of the computer architecture is obtained.

5 Case Study

In this section a case study applying the agglomeration strategy presented in Sect. 4 it is described.

Symmetric and positive defined matrices are very special and they appear very frequent in some scientific applications. A special factorization method called Cholesky decomposition, which is two times faster than other alternatives to solve linear system equations, can be used with this kind of matrices [16].

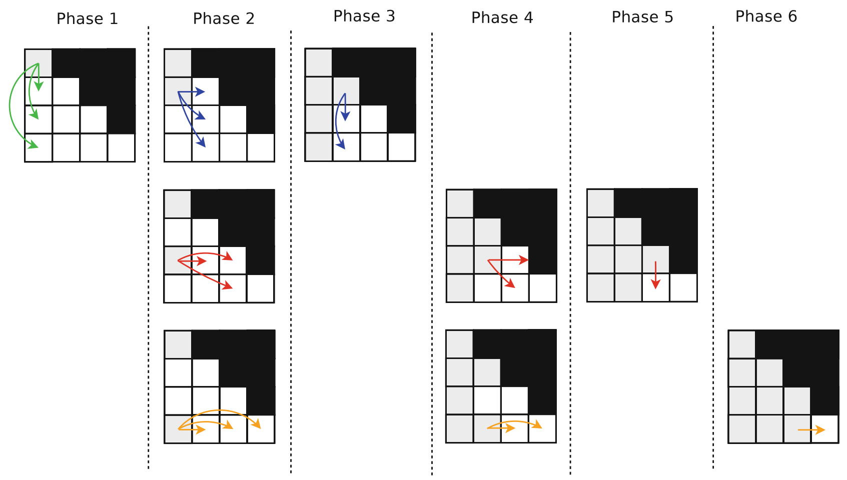

Fig. 3. Data dependency for the Cholesky decomposition on a 4×4 matrix. Gray cells denote an element already computed. The arrows denote a data transmission.

Figure 3 shows the data dependency to compute the Cholesky decomposition on a 4×4 matrix. Due to the symmetry of this matrix, it is enough to work only with the inferior side of the matrix.

Particularly, in this instance the problem is decomposed by cells, meaning that every cell of the matrix represents a parallel process. The arrows between cells become communications between processes needed to transfer data units.

Let Ch be the parallel system that computes the Cholesky decomposition. The software graph depicted in Fig. 4 represents the structure of Ch. The amount of data transferred over each edge is one unit, while the total amount of communications of the parallel system is 20 units.

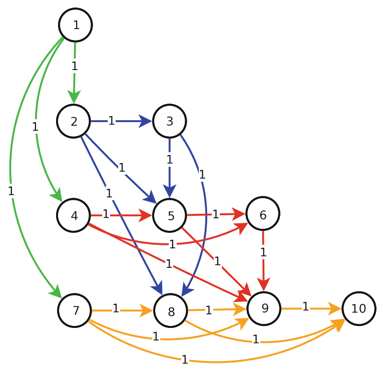

Fig. 4. Software graph for a cell decomposition on a 4×4 matrix.

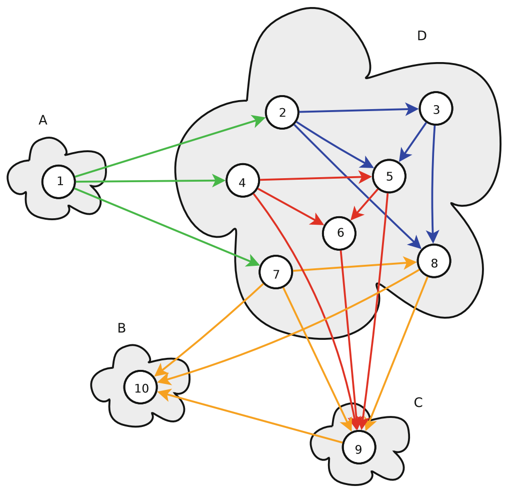

Fig. 5. Agglomeration of the network flow of the Cholesky decomposition problem.

Based on the vertex classification presented in Sect. 4.2, is easy to see that vertex 1 is the only flow generator vertex, while the vertex 10 is the only end vertex. So that, for this example it is not necessary to add the special vertices s and t.

For the agglomeration stage, consider the execution of the parallel system on a *cluster* of $k = 4$ distributed nodes. The resulting agglomeration, obtained by the Algorithm 1, is shown in Fig. 5.

The local and remote communication costs of the agglomeration are:

Partition A: $c_l = 0$, $c_r = 3$.
Partition B: $c_l = 0$, $c_r = 0$.
Partition C: $c_l = 0$, $c_r = 1$.
Partition D: $c_l = 9$, $c_r = 7$.

Such that, the communication time, induced by the agglomeration, of the parallel system Ch is:

$$CT(Ch) = 9 * t_{constant} + 11 * t_{variable}. \tag{14}$$

6 Conclusions

The performance of a parallel system is inherently affected by communication between processes. The communication time added to the execution time of a system is proportional to the amount of data exchanged by the parallel system and the type of communication that it implements.

In general, communication cost through shared memory is less expensive than communication cost via a network medium. Therefore, it is necessary to maximize the amount of local communications and to minimize (to the possible extent) the amount of remote communications, to control the total communication time and mitigate the impact of communications over the execution time of the parallel system.

References

1. Foster, I.: Design and Building Parallel Programs v1.3: An Online Publishing Project (1995)
2. Bokhar, S.: On the mapping problem. IEEE Trans. Comput. **c–30**(3), 207–214 (1981)
3. Kleinberg, J., Tardos, E.: Algorithm Design. Pearson-Addison Wesley, Boston (2005)
4. Chandy, M., Taylor, S.: An Introduction to Parallel Programming: Part II Parallel Program Design. Jones and Bartlett, Burlington (1992). Chap. 7
5. Culler, D., Singh, J., Gupta, A., Kaufmann, K.: parallel computer arquitecture, a hardware / software approach. In: Parallel Programs (1997). Chap. 2
6. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill, Cambridge (2001). Sect. 26.2: The Ford-Fulkerson method
7. Blaise, B.: Introduction to Parallel Computing. Lawrence Livermore National Laboratory (2012). Accessed 11 Sept. 2007
8. Steven, S.: Sorting and Searching. The Algorithm Design Manual, 2nd edn. Springer, London (2008)
9. Bondy, J., Murty, U.S.R.: Graph Theory. Springer, London (2008)
10. Garey, M., Johnson, D.: Computers and Intractability. Bell Telephone Laboratories, A Guide of the Theory of NP-Completeness (1979)
11. Vazirani, V.V.: Approximation Algorithms. Springer, New York (2001)
12. Andreev, K., Racke, H.: Balanced graph partitioning. Theor. Comput. Syst. **39**(6), 929–939 (2006)
13. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Syst. Tech. J. **49**, 291–307 (1970). doi:10.1002/j.1538-7305.1970.tb01770.x
14. Fiduccia, M.: A linear-time heuristic for improving network partitions. In: 19th Design Automation Conference (1982)
15. Chung, F.R.K.: Spectral Graph Theory, vol. 92. Am. Math. Soc. (1997)
16. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C, The art of Scientific Computing, 2nd edn. Cambridge University Press, Cambridge (1992)