Parallel Greedy Graph Matching using an Edge Partitioning Approach

Md. Mostofa Ali Patwary
Department of Informatics, University of Bergen, Norway
Mostofa.Patwary@ii.uib.no

Rob H. Bisseling
Department of Mathematics, Utrecht University, the Netherlands
R.H.Bisseling@uu.nl

Fredrik Manne
Department of Informatics, University of Bergen, Norway
fredrikm@ii.uib.no

Abstract

We present a parallel version of the Karp-Sipser graph matching heuristic for the maximum cardinality problem. It is bulk-synchronous, separating computation and communication, and uses an edge-based partitioning of the graph, translated from a two-dimensional partitioning of the corresponding adjacency matrix. It is shown that the communication volume of Karp–Sipser graph matching is proportional to that of parallel sparse matrix–vector multiplication (SpMV), so that efficient partitioners developed for SpMV can be used. The algorithm is presented using a small basic set of 7 message types, which are discussed in detail. Experimental results show that for most matrices, edge-based partitioning is superior to vertex-based partitioning, in terms of both parallel speedup and matching quality. Good speedups are obtained on up to 64 processors.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Algorithms, Performance

Keywords bulk-synchronous parallel, graph, heuristics, Karp-Sipser, matching, partitioning, sparse matrix

1. Introduction

Let $G = (V, E)$ be a graph with vertex set V and edge set E. A *matching* $M \subseteq E$ is a pairing of adjacent vertices such that each vertex is matched with at most one other vertex. The objective of maximum cardinality matching is to match as many vertices as possible. In this paper, we investigate the parallelization of one particular algorithm for maximum cardinality matching, the KARP–SIPSER algorithm [10], which is a heuristic that has been shown in practice to yield high-quality matchings quickly [16]. Heuristic matching algorithms are often the common choice in practical applications as they are much faster for large problem sizes than optimal algorithms, and because they are easier to implement and parallelize. For bipartite graphs, it has been shown [13] that KARP–SIPSER outperforms other heuristic algorithms such as minimum-degree matching. This motivates our choice to parallelize the KARP–SIPSER algorithm.

We view the graph $G = (V, E)$ as an adjacency matrix A of size $n \times n$ where $n = |V|$ and where for each edge $(i, j) \in E$, A has two nonzeros a_{ij} and a_{ji} such that $a_{ij} = a_{ji}$. In matrix language, our problem then is to find a matching of maximum cardinality among the rows. Since the problem is unweighted, we assume that the numerical value $a_{ij} = 1$ for all nonzeros. The adjacency list of a vertex $i \in V$ is equivalent to row i of A and column i of A. We note that A is symmetric and $a_{ii} = 0$ for all $i = 1, 2, \ldots, n$. (We number vertices/rows from 1 onwards.) In sparse matrix computations, it has been customary already for many years to view matrices for certain purposes as graphs, see e.g. [8, Ch. 1], but in this paper we will exploit the reverse connection, by viewing a graph as a matrix to benefit from sparse matrix partitioning methodology for the purpose of parallelizing a graph algorithm.

Since row i is identical to column i, instead of mentioning row and column i separately, we sometimes call them together *rowcol* i. We maintain only one adjacency list for both of them. The list

HLPP'10, September 25, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0254-8/10/09…$10.00

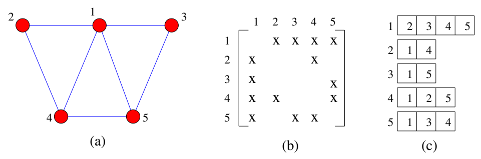

Figure 1. *cage3* matrix of size 5×5 [7]. (a) The graph representation, (b) the corresponding adjacency matrix where each x represents a nonzero, and (c) the rowcols $\{1, 2, \ldots, 5\}$.

contains all entries $\{a_{ij} : 1 \leq j \leq n \text{ and } a_{ij} \neq 0\}$. We get the nonzeros of row i or column i by accessing the entries in rowcol i. Figures $1(a)$ and $1(b)$ show the transformation of a graph to an adjacency matrix and Figure $1(c)$ shows the corresponding rowcols, which can also be seen as adjacency lists.

Our parallelization of the KARP–SIPSER algorithm will be done in bulk-synchronous parallel (BSP) style [19] (see also [3, Ch. 1]), which is characterized by alternating between computation phases and communication phases, each ended by a global barrier synchronization; these phases are commonly called *supersteps*. A computation phase uses only locally available data and can last as long as there is something to compute locally, but it can also be terminated earlier, for instance after a fixed amount of work. This enhances load balancing by detecting at an earlier stage that a processor has run out of work. The BSP style gives a high-level framework for algorithmic development, which eases parallelization of irregular algorithms such as graph algorithms. Examples where this style has been employed are graph coloring [4], edge-weighted graph matching [15], and single-source shortest paths [14].

An advantage of using BSP at a programming level is that the BSPlib communication library [9] takes some of the tediousness away of message-passing for irregular computations; in particular, the bulk-synchronous message passing primitive `bsp_send` is helpful, as it allows sending data to an arbitrary processor without the need for a corresponding receive request. The data is sent to a remote buffer, which can be emptied at the next superstep. BSPlib is available on almost all computer architectures, through a library called BSPonMPI [18], which can be linked to the program, thus effectively turning it into an MPI program. Another advantage of using BSP is that many communication optimizations can be left to the system; for instance, different messages to the same destination are automatically detected and combined. One goal of this paper is to demonstrate how to parallelize a graph algorithm with a high level of irregularity by using BSP.

Instead of using BSP, we could also use message passing, immediately sending matching data once they become available. To make use of this, for instance to improve the quality of the matching, these data must also be received as soon as possible, requiring frequent polling for incoming messages (assuming communication is nonblocking). This would increase communication time and incur more message latency costs, and it would remove the global notion of time that is given by the supersteps of the BSP model.

A fundamental question when parallelizing an algorithm for a distributed-memory computer is how to distribute the data among the processors. A common approach for graph algorithms has been to partition the vertices and then assign each resulting part to a processor (assigning the edges in a corresponding manner). In matrix terms, this leads to a one-dimensional row distribution. Often, the graph has been partitioned beforehand using software such as Metis [11] or Scotch [17]. This has also been the approach taken in our previous work [15].

An alternative approach would be to partition the edges instead of the vertices. In matrix terms, this leads to a general two-dimensional distribution, with in principle a larger space of possible solutions. For parallel sparse matrix–vector multiplication, this is known to lead to much lower communication volumes for certain types of matrices such as web-link matrices, originating outside the traditional application area of Finite Element Methods, see [5, 20]. These matrices from nontraditional areas often have rows and columns with widely varying numbers of nonzeros. One-dimensional methods avoid communication in one direction, but often pay a heavy price in the other direction, especially for relatively dense rows or columns. Two-dimensional methods are able to handle these much better. For graph algorithms, as far as we know, two-dimensional methods have not been employed yet. One of our goals is thus to investigate whether the edge-partitioning approach yields similar benefits as in the matrix–vector case.

We now define some notations that we use throughout the paper. The number of nonzeros in a row i of A is denoted by nz_i. We call a row i a *singleton* if $nz_i = 1$. We let the matching algorithm use p processors denoted by $P_0, P_1, \ldots, P_{p-1}$. We use the two-dimensional partitioning approach of the Mondriaan package [20] to distribute A symmetrically among p processors before the matching starts. This means that a_{ij} and a_{ji} are assigned to the same processor. The nonzeros of a row i could be distributed among several processors, say, q_i processors. Let $lc_{i,s}$ denote the local number of nonzeros of row i in P_s. We choose one of the q_i processors as the *owner* of i, denoted by $P(i)$, and the other $q_i - 1$ processors as the *nonowners* of i, given by the set $nonOwners(i)$, where each nonowner is denoted by $P'(i)$. Note that $P(i)$ stores

$nonOwners(i)$ and nz_i, whereas each $P'(i)$ knows only about $P(i)$. Both the owner and the nonowners maintain their value of $lc_{i,s}$. We use $isMatched(i)$ for the status of matching (either *true* or *false*) and $m(i)$ for the matching partner; both are stored at $P(i)$.

The remainder of this paper is organized as follows. In Section 2, we present the sequential and parallel KARP–SIPSER algorithm and analyze the communication requirements of the parallel algorithm. In Section 3, we describe our experimental methodology, test set details, and our results. We conclude in Section 4.

2. Matching Algorithms

2.1 The Sequential KARP–SIPSER Algorithm

Algorithm 1 Sequential KARP–SIPSER (A)

```
 1: $M \leftarrow \emptyset$
 2: while $A \neq \emptyset$ do
 3:     if $A$ has singleton rows then
 4:         Pick a singleton row $i$ uniformly at random
 5:         Let $a_{ij}$ be the nonzero entry in row $i$
 6:     else
 7:         Pick a nonzero entry $a_{ij}$ uniformly at random
 8:     $M \leftarrow M \cup \{(i,j)\}$
 9:     $A \leftarrow A \setminus (\{a_{i*}\} \cup \{a_{*i}\} \cup \{a_{j*}\} \cup \{a_{*j}\})$
10: return $M$
```

The KARP–SIPSER algorithm [10] is a simple greedy algorithm for maximum cardinality graph matching. We will express it in terms of its adjacency matrix formulation. The idea of the algorithm is as follows. Let A be a symmetric matrix and M the set of matches. If the current matrix A has singleton rows, then the algorithm randomly chooses one such row i and adds (i, j) to the matching M, where a_{ij} is the unique nonzero entry in row i, and removes all the entries from rowcols i and j, and then continues. If the current matrix has more than one entry in each row, hence has no singleton rows, then it picks a random entry a_{ij}, adds (i, j) to the matching and deletes all the entries from rowcols i and j, and then continues. The algorithm stops when A has become empty. Algorithm 1 gives the formal description of the sequential KARP–SIPSER algorithm. Note that while executing the algorithm, the deletion of rowcols generates new singleton rows.

There are two phases in the execution of the KARP–SIPSER algorithm. The first phase starts at the beginning of the whole algorithm and ends when the current matrix has more than one entry in each row. Phase two is the remainder of the algorithm. We note that if M_1 is the set of entries chosen in phase one, then there still exists some maximum cardinality matching that contains M_1, see [1, Fact 1]. Thus the algorithm may have reduced the number of optimal solutions that it can find, but there still is at least one. Furthermore, it has been shown that almost all the remaining rows are matched by the KARP–SIPSER algorithm in the special case where A is a random matrix [1, 6].

2.2 The Parallel KARP–SIPSER Algorithm

In the remainder, we present our parallel implementation of the KARP–SIPSER algorithm. As stated, the algorithm starts with the non-zero entries distributed among the processors and for each row i there is one dedicated owner of that row. Each processor then operates in synchronized rounds where it first performs a local version of the sequential algorithm followed by communication. Here, the match for a singleton row is performed at the processor that has the only (remaining) entry of that row.

In the sequential part, a processor P_s will try to match a predefined number, TpR, of its remaining unmatched rows. Priority is given to singleton rows but if P_s runs out of them before having performed TpR matching attempts, it will try to match some of its remaining rows with random neighbors. This is continued until TpR matching attempts have been reached or until P_s has run out of available rows. In our program texts, P_s will always denote the current processor which will execute the statements of the text.

To see how the algorithm differs from the sequential one, consider when P_s wants to match row i (which it owns) with row j. If P_s also owns row j it can immediately perform the match, but if another processor P_z owns it, then P_s must send a matching request to P_z. Depending on the outcome of this request the match will succeed or fail. Note that the only reason why a matching request could fail is if there were multiple requests to match with the same row in the same or the previous round. For termination, the algorithm relies on some random requests succeeding, which works well in practice. We could also have implemented a stricter mechanism to guarantee termination (e.g. by only requesting matches with higher numbered rows).

In addition to attempting to match its own rows, a processor must also process and answer incoming requests following the communication stage. The overall structure is outlined in Algorithm 2. In the algorithm, Q_s is a queue containing all singleton rows on processor P_s, while $StpR$ and RpR denote the number of attempts per round to perform singleton and random matches.

Algorithm 2 PARALLEL KARP–SIPSER ()

```
1: while $A \neq \emptyset$ do
2:    PROCESS-MESSAGES()
3:    $StpR \leftarrow 0, RdpR \leftarrow 0$
4:    while $StpR + RdpR < TpR$ and $A \neq \emptyset$ do
5:       if $Q_s \neq \emptyset$ then
6:          PICK-SINGLETON-ROW()
7:       else
8:          PICK-RANDOM-ROW()
9:    BSP-SYNC()
```

Table 1. Summary of message types used.

Type	Call	Meaning
Singleton request	$smr(i, j, P_z)$	Matches singleton row i to j
Random request	$rmr(i, j, P_z)$	Matches random row i to j
Confirmation	$cf(i, P_z)$	Confirms success of matching i
Unavailability	$u(i)$	Removes all nonzeros in rowcol i
Handover	$h(i)$	Hands over row i to a nonowner
Give-up	$g(i, P_z)$	Removes P_z from $nonOwners(i)$
Criticality	$ct(i, P_z))$	Local count of row i became 1

2.2.1 The Different Message Types

Our algorithm relies on different types of messages to exchange information between the processors. The different types are summarized in Table 1 and explained in this subsection.

The first type of message is a *singleton match request*. Suppose processor P_s wants a singleton row i to match with row j, but $P(j) \neq P_s$. Therefore, P_s must send a message to $P(j)$ requesting to match j with i. We use smr to denote such a match request. Since $P(j)$ could receive several match requests from several processors and $P(j)$ can match j only with one i, $P(j)$ sends back reply messages, called *confirmation message* (denoted by cf), to update the requesters about the success of the match request. (If a requester does not receive a confirmation back within two rounds, this means that the request has failed.) The third type of message is the *unavailability message* (denoted by u). When a row i is matched with a row j, we need to remove all the entries from row i, column i, row j, and column j. Since we have only one adjacency list called rowcol i for each row i and column i, to remove all entries from both of them, it is sufficient to remove the entries from rowcol i. Therefore, we remove rowcol i and j. We first remove the entries in rowcol i and j from P_s, and then send unavailability messages to the other $q_i - 1$ and $q_j - 1$ processors holding row i and row j, respectively, to remove the entries in rowcol i and j from them. There is another type of match request, called *random match request* (denoted by rmr), used to match a row i with row j, given that $P(i) = P_s$. This type of message is only initiated when P_s can perform more work in the current round but $Q_s = \emptyset$, where Q_s denotes the queue of singleton rows in P_s.

We now discuss the remaining three types of messages. Consider a situation where i is a singleton row and the local count for row i in $P(i)$ is 0. Then, $P(i)$ must send a message to $P'(i)$, the only nonowner of i, to insert i into its singleton queue. We call this message a *handover message*, denoted by h. The next type of message is called *give-up message*, denoted by g, which is always sent from a nonowner, $P'(i)$ of i, to $P(i)$ to remove $P'(i)$ from $nonOwners(i)$. Processor $P'(i)$ sends such a message when its local count for row i reduces to 0. The last type of message, *criticality message* (denoted by ct), is sent from a nonowner $P'(i)$ to $P(i)$. We use this message to update $P(i)$ that the local count for row i in $P'(i)$ has been reduced to 1. This message enables the owner $P(i)$ to verify whether i has become a singleton row. The criticality message(s) for row i together with the knowledge of nz_i enable the owner of row i at the earliest possible moment to detect that a locally empty row has become singleton and thus to insert it into the appropriate queue on the only nonempty processor by using a handover message. We could have decided not to use criticality messages. Then, we would need a mechanism for transferring ownership, which is more complicated and would involve extra communication.

2.2.2 The Functions

PROCESS-MESSAGES **function**: This function processes all the incoming messages. We do the following, based on the message type. For each singleton match request $smr(i, j, P_z)$ received from processor P_z, we call MATCH-ROWCOL$(i, j, P_z, singleton)$ to match i with j. For each unavailability message $u(i)$, we call REMOVE-ROWCOL(i) to remove i from P_s. For each handover message $h(i)$, we check whether i is singleton. If so, we push i into the singleton queue, Q_s. For a criticality message $ct(i, P_z)$, we reduce the nonzero count of row i, nz_i, by $lc_{i,z} - 1$. We also check whether this reduction makes i a singleton. If so, we send a handover message $h(i)$ to P_z to push i into its singleton queue. For each confirmation message $cf(i, P_z)$, we call CONFIRM(i, P_z) to remove row i. For each random match request $rmr(i, j, P_z)$, we call MATCH-ROWCOL$(i, j, P_z, random)$

to match j with i. For each give-up message $g(i, P_z)$, we remove P_z from $nonOwners(i)$. Since processor P_z sends the give-up message only when it removes its last local entry in row i, we reduce the nonzero count of row i, nz_i, by 1 and if relevant, insert row i into the singleton queue. We do this by calling DECREMENT(i).

In our implementation, all messages types have the same priority, and they are processed in the order they were entered into the receive buffer of the BSP system. It is possible, however, to sort them by type first, e.g., to give preference to singleton match requests over random match requests.

Algorithm 3 PICK-SINGLETON-ROW() - Picks singleton rows and matches them.

```
1: $i \leftarrow Q_s.pop()$
2: **if** $lc_{i,s} = 1$ **then**
3:     $lc_{i,s} \leftarrow 0$
4:     Let $a_{ij}$ be the entry in row $i$
5:     REMOVE-ROWCOL$(j)$
6:     **if** $P(j) = P_s$ **then**
7:         MATCH-ROWCOL$(i, j, P_s, singleton)$
8:     **else**
9:         send a singleton match request $smr(i, j, P_s)$ to $P(j)$
10:    $StpR \leftarrow StpR + 1$
```

PICK-SINGLETON-ROW **function** (given by Algorithm 3). The goal here is to pick and match a singleton row. The function first pops a row i from Q_s and then verifies whether i is still a singleton row by checking if $lc_{i,s} = 1$. (This check is necessary because unavailable rows are not removed from queues.) If not, it continues to the next singleton row in Q_s. If the answer is yes, it does as follows. Let a_{ij} be the unique entry in row i. Although j is the only option for i to match with, j could have several such singleton candidates and it can match only with one of them. Now, irrespective of which singleton row it is matching with, all entries from row j must be removed, because j will match with this i or one of the other candidate singleton rows, which eventually leads to the removal of rowcols i and j. So we can safely remove all entries from row j in P_s by calling REMOVE-ROWCOL(j). We also remove the only entry a_{ij} in row i, by setting $lc_{i,s} \leftarrow 0$. The next step is to check where the owner of j, $P(j)$ is. If $P(j) = P_s$, we call MATCH-ROWCOL$(i, j, P_s, singleton)$ immediately to match j with i. Otherwise, we send a singleton match request $smr(i, j, P_s)$ to $P(j)$. The parameters P_s and $singleton$ of MATCH-ROWCOL mean that P_s has invoked the function and the matching request is of singleton type.

PICK-RANDOM-ROW **function**: This is similar to PICK-SINGLETON-ROW except that it picks a random row i owned by this processor to match with a random neighbor j. If there are multiple choices for j then priority is first given to local neighbors. If no local neighbor exists, a random match request is sent to $P(j)$.

Algorithm 4 MATCH-ROWCOL$(i, j, P_z, type)$ - Matches j with i, updates $P_z = P(i)$ about success, and removes all nonzeros from rowcols i and j. Can only be called by $P_s = P(j)$.

```
1: **if** $isMatched(j) = false$ **then**
2:     $m(j) \leftarrow i, isMatched(j) \leftarrow true, nz_j \leftarrow 0$
3:     **if** $type = random$ **then**
4:         **if** $P_z = P_s$ **then**
5:             CONFIRM$(i, P_s)$
6:         **else**
7:             REMOVE-ROWCOL$(i)$
8:             send a confirmation message $cf(i, P_s)$ to $P_z$
9:     **if** $P_z \neq P_s$ **then**
10:        REMOVE-ROWCOL$(j)$
11:    send unavailability $u(j)$ to each $P'(j) \in nonOwners(j)$
```

MATCH-ROWCOL **function** (given by Algorithm 4). The goal here is to match row j with row i if possible and take necessary actions if the matching is successful. We first verify whether j has already been matched by checking $isMatched(j)$. The next step is to check the *type* of the matching. If the type is singleton, we do not need to give any confirmation back to the owner $P(i) = P_z$, because its only remaining job was to remove the unique entry a_{ij} in row i, which has already been done. If the type is random, we need to send a confirmation back to $P(i)$ to let it remove row i. If $P_z = P_s$, we call CONFIRM(i, P_s) to remove row i from P_z and $nonOwners(i)$. If $P_z \neq P_s$, we first remove row i from P_s and then send a confirmation message $cf(i, P_s)$. The next step is to remove rowcol j from $P_s = P(j)$ and $nonOwners(j)$. Note that j has been removed from processor P_z in case of a singleton match request, and it will be removed following the confirmation in case of a random match request. If $P_z \neq P_s$, we remove rowcol j from P_s locally. We then send unavailability messages $u(j)$ to all the other $q_j - 1$ or $q_j - 2$ nonowners. We set $m(j) = i$, as this can be done locally by $P(j)$, but we do not set $m(i) = j$ as this would require communication with $P(i)$, which would be unnecessary since rowcol i will be removed immediately afterwards and hence cannot be matched anymore. No redundant matching information is thus communicated or stored.

Algorithm 5 REMOVE-ROWCOL(i) - Removes all entries from rowcol i in P_s.

```
1: while $lc_{i,s} > 0$ do
2:     Let $a_{ij}$ be the last entry in rowcol $i$
3:     swap $a_{ji}$ with the last entry in rowcol $j$
4:     $lc_{j,s} \leftarrow lc_{j,s} - 1$
5:     if $P(j) = P_s$ then
6:         DECREMENT($j$)
7:     else if $lc_{j,s} = 1$ then
8:         send a criticality message $ct(j, P_s)$ to $P(j)$
9:     else if $lc_{j,s} = 0$ then
10:        send a give-up message $g(j, P_s)$ to $P(j)$
11:    $lc_{i,s} \leftarrow lc_{i,s} - 1$
```

REMOVE-ROWCOL **function** (given by Algorithm 5). This function removes all entries from rowcol i in P_s. At every iteration of the while-loop, it picks the last entry a_{ij} from the adjacency list of row i. We remove a_{ij} from the adjacency list of rowcol i by reducing the local count $lc_{i,s}$ by 1. Since the matrix is symmetric, we also have to remove a_{ji} from rowcol j. We do this by swapping a_{ji} with the last entry of the adjacency list of rowcol j and reducing the local count $lc_{j,s}$ by 1. The swap and reduction operations make the removal efficient. Since j has not been matched yet, we consider whether the removal of a_{ji} creates any of the following three cases. The first case is where P_s owns j, so that we can safely reduce the nonzero count of row j by 1 and insert j into the singleton queue if possible. We do this by calling DECREMENT(j). The second case is where the removal of a_{ji} reduces the local count $lc_{j,s}$ to 1, so that we have to send a criticality message $ct(j, P_s)$ to $P(j)$ to reduce the nonzero count nz_j by $lc_{j,s} - 1$, where $lc_{j,s}1$ is the initial local count. The third case is where $lc_{j,s} = 0$, so that P_s does not have any entry in row j anymore and we can send a give-up message $g(j, P_s)$ to $P(j)$ to remove P_s from $nonOwners(j)$.

CONFIRM **function**: The goal here is to remove all entries in row i from $P(i)$ and $nonOwners(i)$. We remove row i from $P(i)$ by calling REMOVE-ROWCOL(i) and from the nonowners by sending unavailability messages $u(i)$ to all of them, except to a processor P_z that previously sent a confirmation message to $P(i)$ causing this function to be called.

DECREMENT **function**: This function first decrements the nonzero count nz_i of row i. It then checks whether this turns row i into a singleton row. If so, it looks where the last remaining entry a_{ij} of row i is. If a_{ij} is local, that is, $lc_{i,s} = 1$, we push i into the singleton queue Q_s. Otherwise, we send a handover message $h(i)$ to the only nonowner of i, $P'(i)$, asking $P'(i)$ to push i into its singleton queue.

2.3 Communication Requirements

Following the BSP model [3, Ch. 1] for our parallel matching algorithm, we separate computation and communication into distinct, rather than intermingled, stages. The parallelism in the computation is obtained from the assumption that each processor will have a large number of local matches to perform between the communication supersteps. This allows us to analyse the computation and communication requirements separately. The computation part will be studied experimentally in the next section. For the communication part, we can obtain theoretical bounds on the total communication volume of the algorithm, as follows.

We analyse the communication requirements by considering a row i, with q_i processors. We assume that $q_i \geq 1$, because we can remove empty rows and columns. Let j be the requested matching partner of i. We distinguish between requests that succeed and those that fail. We will examine what the current processor P_s needs to communicate for row i. We count each message as one data word.

First, consider the case where i is a singleton row, $i \in Q_s$, see Algorithm 3. It does not matter here whether or not $P_s = P(i)$.

- Case $P_s \neq P(j)$: P_s sends one message (a matching request) to $P(j)$. If i succeeds to match with j, then $P(j)$ sends $q_j - 2$ messages asking for removal of rowcol j to the nonowners of j, except P_s which initiated the matching request and therefore already removed rowcol j. If i fails to match with j, then $P(j)$ does not send any message at all. The total number of messages is $q_j - 1$ for success and 1 for failure.
- Case $P_s = P(j)$: as the previous case, but no matching request needs to be sent, and the number of removal requests in case of success is $q_j - 1$. The total number of messages is $q_j - 1$ for success and 0 for failure.

Therefore, for each singleton row i, the communication volume is at most $q_j - 1$. A similar analysis yields that for each randomly picked row, the volume is at most $q_i + q_j - 2$.

For the other three types of messages summarized in Table 1, row i incurs at most one handover message, and $q_i - 1$ give-up messages during the whole algorithm. But if $P(i)$ sends a handover message to $P'(i)$, then $P'(i)$ will never send a give-up message to $P(i)$, and vice versa. Therefore for each row i we get at most $q_i - 1$

give-up and handover messages, and $q_i - 1$ critical messages, with a total upper bound of $2q_i - 2$.

We can now add all the communication bounds. Let s be the number of matchings that involve at least one singleton row. Since the matrix A has n rows, the number of matched rows picked randomly is at most $\frac{n-2s}{2} = \frac{n}{2} - s$. Without loss of generality we renumber the rows, so that the matched singleton rows come first, and the matched random rows second, so they are in the range $1 \leq i \leq n/2$, and we also take care that their matches are in the second half, $n/2 + 1 \leq m(i) \leq n$. Assume for a moment that all match requests succeed. An upper bound for the total communication volume is then

$$\begin{aligned} &Vol(Matching) \\ \leq\ & \sum_{i=1}^{n/2} q_{m(i)} + \sum_{i=s+1}^{n/2} q_i + 2\sum_{i=1}^{n} q_i - 3n + s \\ \leq\ & \sum_{i=1}^{n/2} q_{m(i)} + \sum_{i=1}^{n/2} q_i + 2\sum_{i=1}^{n} q_i - 3n \\ =\ & 3\sum_{i=1}^{n} (q_i - 1) = \frac{3}{2} \cdot Vol(SpMV). \end{aligned}$$

Here, we express the upper bound in terms of sparse matrix–vector multiplication, which has a volume of $Vol(SpMV) = 2\sum_{i=1}^{n} (q_i - 1)$ for a symmetrically partitioned matrix. This is useful because the SpMV kernel is important and many partitioning algorithms and software packages exist that minimize its volume.

Now we drop the non-failure assumption. For singleton rows, at most one data word is sent and the row is removed and remains unmatched. The upper bound then still holds. For randomly picked rows, the situation is more complicated. If the request fails and $P(i) \neq P(j)$, this incurs one message. In principle, the number of such failures is unbounded, since randomly picked rows can be tried again, but in practice the volume will be limited as preference is given to local matches (not causing communication in case of failure) and the penalty in the non-local case is only one communication. This will add a number R of failed random requests to the upper bound.

A lower bound on the communication can be obtained as follows. Assume the best case, where all matches are local. For each row i, we need $q_i - 1$ messages to remove it. This leads to

$$Vol(Matching) \geq \sum_{i=1}^{n} (q_i - 1) = \frac{1}{2} \cdot Vol(SpMV).$$

Table 2. Benchmarked BSP parameters for the IBM pSeries 575.

p	r	(flop)		(μs)	
	(Gflop/s)	g	l	g	l
1	1.343	355	3,926	0.26	2.92
2	1.336	358	15,681	0.27	11.73
4	1.338	384	27,543	0.29	20.58
8	1.340	384	56,875	0.29	42.44
16	1.334	366	124,430	0.27	93.30
32	1.329	417	268,559	0.31	202.10
64	1.339	408	717,400	0.30	535.74

3. Experimental Results

3.1 Experimental Setup

We performed experiments on *Huygens*, an IBM pSeries 575 supercomputer at SARA in Amsterdam, consisting of 104 nodes, each with 16 processors and 128 GByte of memory. Each processor is an IBM Power6 dual-core 4.7 GHz processor where each core has 128 kByte of L1 cache and 4 MByte of L2 cache. Each processor has 32 MByte of L3 cache. The machine is running Linux (kernel version 2.6.27.45-0.1.2-ppc64). All algorithms were implemented in C++ using the BSPonMPI library (version 0.3) [18] and compiled with the IBM XL C/C++ compiler (version 10.01.0000.0002) using the -O3 optimization level.

We obtained the BSP parameters of the system by BSP benchmarking [3, Ch. 1] for a given number of processors p, as shown in Table 2. These parameters are r, the single-processor computing rate in Gflop/s, g, the time taken by one processor to send or receive one data word, and l, the time taken to synchronize all processors.

We use four test sets of matrices. Test sets 1 and 2 consist of 10 real-world symmetric matrices and four real-world unsymmetric square matrices, respectively, of varying sizes drawn from different application areas such as medical science, structural engineering, civil engineering, circuit simulation, electrical engineering, DNA electrophoresis, information retrieval, and the automotive industry [7, 12]. Test set 3 includes three synthetic small-world matrices and test set 4 contains three synthetic Erdös-Rényi style random square matrices generated by the GTGraph package [2]. For convenience, we label the test sets rw, sw, and er, for real-world, small-world, and Erdös-Rényi, respectively. The matrices from test sets 2–4 were made symmetric by adding A and A^T. All diagonal entries were removed from the matrices.

The structural properties of the test matrices are given in Table 3. The columns are the labels, number of rows, number of nonzeros, average and maximum number of nonzeros per row. The names

Table 3. Structural properties of the input matrices.

	n	nz	nz/n avg	nz/n max		n	nz	nz/n avg	nz/n max
rw1	999,999	3,995,992	3	4	rw11	281,903	3,985,272	14	38,625
rw2	1,585,478	6,075,348	3	5	rw12	16,783	9,306,644	554	14,671
rw3	52,804	10,561,406	200	2,702	rw13	683,446	13,269,352	19	83,470
rw4	2,063,494	12,964,640	6	95	rw14	343,791	26,493,322	77	434
rw5	63,838	14,085,020	220	3,422	sw1	50,000	14,112,206	282	5,096
rw6	504,855	17,084,020	33	39	sw2	75,000	24,466,808	326	6,273
rw7	503,712	36,312,630	72	842	sw3	100,000	33,727,170	337	7,989
rw8	952,203	45,570,272	47	76	er1	100,000	3,319,658	33	59
rw9	1,508,065	51,164,260	33	34	er2	150,000	6,753,302	45	76
rw10	914,898	54,553,524	59	80	er3	200,000	12,008,022	60	100

Table 4. Communication volume in 1000 words for $p = 32$.

	$SpMV$		$Matching$			$SpMV$		$Matching$	
$Name$	$1D$	$2D$	$1D$	$2D$	$Name$	$1D$	$2D$	$1D$	$2D$
rw1 (ecology2)	53	51	60	55	rw11 (Stanford)	340	141	479	234
rw2 (G3_circuit)	81	65	92	73	rw12 (gupta3)	710	44	1,305	61
rw3 (crankseg_1)	78	78	155	152	rw13 (St_Berk.)	716	448	1,152	812
rw4 (kkt_power)	118	120	106	107	rw14 (F1)	139	130	148	139
rw5 (crankseg_2)	92	90	181	171	sw1	1,007	417	2,111	303
rw6 (af_shell8)	51	47	85	65	sw2	1,957	829	3,999	563
rw7 (inline_1)	104	105	115	118	sw3	2,017	832	4,255	528
rw8 (ldoor)	131	128	140	148	er1	1,856	1,133	1,788	1,157
rw9 (af_shell10)	113	105	169	150	er2	3,451	1,841	3,721	1,635
rw10 (boneS10)	150	145	228	189	er3	5,476	2,569	6,350	1,990

of the matrices are given in Table 4. To obtain the runtime of an algorithm for a given matrix, we execute the algorithms three times and then take the minimum time, based on the assumption that this timing suffers the least from interference by other use of the hardware resources. In all three runs, the actual computations and hence the quality of the matching are the same (we start with the same random number seeds), so that timing differences between the runs are not due to different amounts of work performed.

3.2 Scalability Experiments

To check how well the edge partitioning approach works, we first compare it with a vertex partitioning, where we can use the same Mondriaan framework. In the vertex partitioning, we simply impose the extra constraint that all nonzeros in a row up to the matrix diagonal are assigned to the same processor. This way, we can view vertex partitioning as a special case of edge partitioning. Table 4 presents the communication volumes of sparse matrix–vector multiplication and matching, both for a vertex (1D) partitioning and an edge (2D) partitioning on 32 processors. The SpMV volume is a direct outcome of the partitioning by the Mondriaan package [20]

Table 5. Speedup as a function of TpR for $p = 32$. Boldface denotes the highest speedup obtained.

$TpR =$	100	200	400	800	1600		100	200	400	800	1600
rw1	0.67	**0.74**	0.62	0.40	0.24	rw11	4.25	5.32	6.15	6.17	**6.45**
rw2	0.66	**0.72**	0.59	0.38	0.20	rw12	25.36	18.99	**30.55**	29.55	30.35
rw3	12.65	13.07	**15.13**	14.53	14.42	rw13	1.18	1.59	1.83	**1.85**	1.73
rw4	**1.55**	1.30	0.72	0.31	0.17	rw14	13.15	16.67	19.54	21.63	**24.23**
rw5	14.11	16.62	19.69	**21.09**	19.99	sw1	29.49	33.38	**34.63**	30.58	30.82
rw6	6.26	9.29	12.92	**14.03**	13.82	sw2	27.87	31.16	33.85	**33.91**	33.75
rw7	9.19	11.17	12.09	12.85	**12.88**	sw3	33.35	40.83	42.18	**44.64**	42.43
rw8	6.93	8.45	9.22	**9.25**	8.83	er1	5.20	6.02	7.64	8.60	**9.51**
rw9	6.44	9.66	12.19	**13.08**	11.50	er2	7.15	9.60	11.00	12.71	**13.63**
rw10	7.07	8.41	**8.82**	7.97	6.60	er3	14.31	15.97	18.14	19.72	**21.55**

(version 2.01) in symmetric mode, where the maximum number of edges per processor is not allowed to exceed the average by more than 3%. The processor with most nonzeros in row i was chosen as the owner $P(i)$, because it is more likely to possess the last remaining nonzero of the row after the other nonzeros have been removed, thus saving a handover message. The volume for matching is the volume measured by counters in the program, which register the number of (integer) data words sent.

Table 4 shows that on 32 processors, the volume for the matching is in a range from 0.63 to 2.11 times the SpMV volume. We also observed a range between 0.63 and 2.18 for 2, 4, 8, 16, 32, and 64 processors. This shows that partitioning for the SpMV objective is also a good optimizer for matching, and possibly for other graph problems as well. The table shows a savings in communication volume of a factor of 2 for small-world and Erdös-Rényi matrices when moving from 1D to 2D, and even larger savings for the real-world matrices from test set 2. Note the large 16-fold decrease for the linear programming matrix rw12 (gupta3). For the symmetric real-world matrices (test set 1), only some modest gains can be observed, but also a few cases with a small loss.

Table 5 gives the speedup of our parallel KARP–SIPSER implementation on 32 processor cores compared to the time of our sequential implementation. We examine the performance as a function of the input parameter TpR, which is the total number of rows processed in a round and which represents the chosen granularity of the computation. Choosing a small value of TpR leads to many rounds in the whole algorithm, and hence many supersteps and synchronizations. For $p = 32$, one synchronization costs about $l = 270,000$ flop time units, see Table 2, so the number of operations carried out per processor in a round should at least be this number. As a rough estimate, for the matrix er3, this means handling about 1500 rows of 60 nonzeros each, with (an estimated)

Table 6. Matching quality (in %) for the experiments of Table 5. Boldface denotes the highest quality obtained.

$TpR =$	100	200	400	800	1600		100	200	400	800	1600
rw1	**98.15**	98.14	98.13	98.08	98.12	rw11	**71.75**	71.61	71.48	71.32	71.11
rw2	**96.71**	96.69	96.61	96.52	96.45	rw12	**98.31**	98.00	97.35	97.35	97.35
rw3	**99.21**	99.15	99.13	99.16	99.19	rw13	**66.19**	66.15	66.09	65.99	65.87
rw4	88.55	**88.58**	**88.58**	88.57	88.57	rw14	**99.54**	99.52	99.53	99.51	99.49
rw5	**99.26**	99.24	99.24	99.20	99.18	sw1	**79.81**	78.07	77.06	75.66	75.59
rw6	**99.93**	**99.93**	99.92	**99.93**	**99.93**	sw2	**90.74**	88.87	86.25	84.09	81.89
rw7	**99.56**	99.55	99.55	99.54	99.53	sw3	**81.87**	80.13	78.47	77.29	76.01
rw8	**98.58**	**98.58**	**98.58**	**98.58**	98.57	er1	**97.50**	93.45	85.67	78.69	74.13
rw9	**99.94**	**99.94**	**99.94**	**99.94**	**99.94**	er2	**98.43**	95.63	89.12	82.54	76.07
rw10	**99.58**	99.56	99.55	99.55	99.55	er3	**95.98**	93.14	88.94	83.42	77.59

three operations per nonzero, where for simplicity we assume that every row is completely assigned to one processor; in reality, some rows are partitioned. The advantage of a small TpR is better load balance: when a processor runs out of work this will be detected earlier, and communications are performed more frequently, thus enabling processors to carry out work that otherwise would have to wait until later.

Finding the right value of TpR is important to get good speedups. Fortunately, the parameter is not very sensitive, and a whole range of values gives the highest obtainable speedup; e.g. for er3, this is the range 400–1600. The overall highest speedup obtained (44.64 for sw3) is superlinear, which must be due to beneficial cache-effects or to the fact that the sequential and parallel algorithms do not perform exactly the same amount of work. (The parallel algorithm may be forced to pick random rows more often than the sequential algorithm thus performing less work and delivering lower quality.) Other problem instances may have benefited from these effects as well.

The choice of TpR also influences the quality of the solution (defined as the ratio between the number of matched rows and the total number of rows) for the matrices from test sets 3 and 4, see Table 6. Here, the quality decreases with increasing TpR. For these matrices, which have high communication volumes due to their random nature, few singleton rows can be processed in a round, forcing the processing of random rows in many cases.

Tables 7 and 8 present the speedups for the vertex and edge partitioning approaches. In all cases, the value of TpR was set at an optimal value based on an empirical parameter search, choosing the value among 100, 200, 400, 800, and 1600 that gave the highest speedup. In general, it can be observed that vertex partitioning and edge partitioning do not differ much in time and quality for test set 1, but that edge partitioning is much faster for test sets 2–4.

Table 7. Speedup (Su) and matching quality in % (Ql) using vertex (1D) partitioning.

	Seq	$p = 2$		$p = 4$		$p = 8$		$p = 16$		$p = 32$		$p = 64$	
	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql
rw1	100.00	0.17	99.84	0.15	98.02	0.29	98.11	0.45	97.88	0.70	97.92	**0.84**	98.09
rw2	99.93	0.12	96.95	0.18	96.62	0.28	96.51	0.44	96.45	0.71	96.40	**0.86**	96.18
rw3	99.59	1.60	99.57	3.49	99.48	5.82	99.46	12.62	99.39	**20.35**	99.16	13.01	99.42
rw4	91.54	0.49	88.09	0.62	88.44	0.79	88.35	1.37	88.48	**1.47**	88.45	1.29	88.42
rw5	99.60	1.78	99.61	3.68	99.56	6.92	99.50	13.56	99.36	**22.90**	99.07	17.58	99.36
rw6	99.99	1.58	99.97	2.80	99.97	4.80	99.96	8.11	99.94	13.28	99.92	**17.29**	99.90
rw7	99.62	1.32	99.58	2.00	99.56	2.98	99.58	6.05	99.57	14.07	99.52	**28.95**	99.48
rw8	98.53	1.30	98.72	2.11	98.74	3.24	98.73	5.38	98.72	9.39	98.73	**13.99**	98.73
rw9	99.99	1.71	99.99	2.90	99.98	5.04	99.97	8.16	99.96	13.99	99.95	**19.73**	99.93
rw10	99.70	1.37	99.67	2.26	99.65	3.56	99.64	5.65	99.62	8.27	99.60	**11.34**	99.58
rw11	74.26	1.00	72.09	1.66	72.02	2.38	71.57	3.98	71.20	**5.46**	70.81	4.61	70.26
rw12	99.06	1.91	73.18	3.07	57.98	4.83	64.03	5.13	82.40	**5.78**	86.68	3.59	97.78
rw13	68.56	0.62	66.19	0.81	66.29	0.89	66.16	1.37	65.94	1.81	66.03	**1.92**	65.08
rw14	99.65	1.49	99.61	2.65	99.60	4.81	99.57	10.54	99.55	20.93	99.49	**33.30**	99.42
sw1	82.77	2.28	82.56	4.64	82.53	8.46	82.39	14.39	82.32	**17.33**	82.04	14.51	79.89
sw2	93.68	2.22	93.33	4.65	93.28	8.73	93.16	15.27	92.87	**21.43**	92.93	20.94	90.27
sw3	82.76	2.28	82.65	5.41	82.47	9.28	82.44	17.19	82.42	26.75	82.37	**27.97**	81.35
er1	99.99	1.32	98.87	1.66	98.66	1.82	97.81	2.35	86.87	3.27	63.63	**5.23**	46.66
er2	99.99	1.48	99.16	2.26	99.18	2.60	99.17	2.90	96.63	3.75	71.99	**5.31**	53.75
er3	100.00	1.47	99.33	2.48	99.39	2.88	99.33	3.38	95.60	3.99	89.41	**5.05**	60.85

The matrix rw12 shows a much larger maximum speedup (30.55) for edge partitioning than for vertex partitioning (5.78), and also a better quality. This holds for most cases, but there are exceptions, cf. sw1–sw3 for $p = 64$. The higher speedups for edge partitioning are primarily caused by the lower communication volume, see Table 4 for $p = 32$, but other factors play a role as well. For instance, the smallest problem rw1 shows no speedup at all, which is most likely caused by severe load imbalance and a relatively large synchronization overhead.

4. Conclusion

In this work, we have demonstrated how a graph matching algorithm, the KARP–SIPSER algorithm, can be parallelized efficiently by viewing it as a sparse matrix algorithm, and by making use of sparse matrix partitioning methodology. A number of conclusions can be drawn:

- Edge-based partitioning gives for certain types of graphs, such as small-world graphs, a large improvement compared to vertex-based partitioning. For other types of matrices, a more modest improvement is obtained. In the remaining few cases, the differences are small.
- Improvements obtained by better partitioning lead to better locality, thus reducing the amount of communication required and hence making the parallel algorithm run faster. They also enable more computations to be done locally within a superstep, keeping work queues filled longer and hence improving the matching quality, i.e., the percentage of matched vertices.

Table 8. Speedup (Su) and matching quality in % (Ql) using edge (2D) partitioning.

	Seq	$p = 2$		$p = 4$		$p = 8$		$p = 16$		$p = 32$		$p = 64$	
	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql	Su	Ql
rw1	100.00	0.18	99.84	0.18	98.79	0.34	98.86	0.53	98.54	0.74	98.14	**0.92**	98.02
rw2	99.93	0.11	96.95	0.17	96.50	0.30	96.83	0.47	96.99	0.72	96.69	**0.83**	96.62
rw3	99.59	1.60	99.59	3.42	99.57	6.49	99.39	12.57	99.12	**15.13**	99.13	10.45	98.92
rw4	91.54	0.49	88.09	0.58	88.15	0.80	88.19	1.31	88.44	**1.55**	88.55	1.36	88.55
rw5	99.60	1.78	99.61	3.49	99.55	6.76	99.47	13.97	99.35	**21.09**	99.20	13.93	98.78
rw6	99.99	1.57	99.98	2.77	99.97	4.68	99.95	8.36	99.95	14.03	99.93	**15.13**	99.89
rw7	99.62	1.24	99.56	1.84	99.58	3.13	99.55	6.10	99.54	12.88	99.53	**25.75**	99.47
rw8	98.53	1.29	98.72	2.11	98.72	3.32	98.74	5.59	98.64	9.25	98.58	**14.01**	98.68
rw9	99.99	1.64	99.99	2.97	99.98	5.20	99.97	9.13	99.96	13.08	99.94	**17.97**	99.93
rw10	99.70	1.37	99.67	2.33	99.62	3.77	99.62	6.04	99.59	8.82	99.55	**11.70**	99.51
rw11	74.26	0.95	71.99	1.63	71.96	2.37	71.73	4.57	71.21	**6.45**	71.11	5.45	70.28
rw12	99.06	2.65	99.35	6.50	99.28	14.23	97.67	25.44	97.96	**30.55**	97.35	23.39	90.98
rw13	68.56	0.65	66.30	0.81	66.31	1.01	67.18	1.31	65.85	1.85	65.99	**2.57**	65.73
rw14	99.65	1.49	99.61	2.79	99.57	5.05	99.57	11.38	99.53	24.23	99.49	**37.74**	99.40
sw1	82.77	2.28	82.56	4.82	82.44	9.36	82.28	19.74	80.31	34.63	77.06	**41.55**	73.33
sw2	93.68	2.20	93.33	4.69	92.94	8.96	92.36	19.10	89.64	33.91	84.09	**55.39**	78.38
sw3	82.76	2.27	82.65	5.23	82.47	9.92	82.01	21.02	79.91	44.64	77.29	**72.16**	73.80
er1	99.99	1.34	98.87	2.27	98.65	4.35	97.43	7.13	83.71	9.51	74.13	**13.14**	58.46
er2	99.99	1.48	99.16	2.98	99.23	5.71	97.89	9.83	86.62	13.63	76.07	**21.12**	64.23
er3	100.00	1.48	99.39	3.10	99.39	5.66	99.15	11.89	92.63	21.55	77.59	**29.96**	67.43

- We have established a theoretical relation between the communication volume of parallel graph matching by the KARP–SIPSER algorithm and sparse matrix-vector multiplication,
  $$\frac{1}{2} \cdot Vol(SpMV) \leq Vol(Matching) \leq \frac{3}{2} \cdot Vol(SpMV) + R$$
  where R represents the number of random match requests that failed during the algorithm. The range we encountered in practice for edge partitioning, is between 0.63 to 1.95 times $Vol(SpMV)$ for 2, 4, 8, 16, 32, and 64 processors.
- We have obtained good speedups for many matrices without compromising the quality of the matching. Up to 16 processors, the matching quality stays constant, see Table 8. After that, for some matrices it decreases as work queues become empty more quickly, thereby forcing random rows to be matched.

For future work, we see the present algorithm as a representative of a whole class for which an edge-based approach will be suitable and a relation with sparse matrix–vector multiplication can be established. We intend to generalize our approach in this direction.

Acknowledgments

This work has been performed under the HPC-Europa2 Transnational Access programme, in FP7, grant agreement nr. 228398.

References

[1] J. Aronson, A. Frieze, and B. G. Pittel. Maximum matchings in sparse random graphs: Karp-Sipser revisited. *Rand. Struct. Alg.*, 12(2):111–177, 1998.

[2] D. A. Bader and K. Madduri. GTGraph: A synthetic graph generator suite. http://www.cc.gatech.edu/~kamesh/GTgraph, 2006.

[3] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI.* Oxford University Press, 2004.

[4] D. Bozdag, A. Gebremedhin, F. Manne, E. Boman, and U. Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. *J. Par. Distr. Comput.*, 68(4):515–535, 2008.

[5] U. V. Catalyurek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proc. IPDPS*, 2001.

[6] P. Chebolu, A. Frieze, and P. Melsted. Finding a maximum matching in a sparse random graph in $O(n)$ expected time. *Proc. ICALP*, pages 161 – 172, 2008. LNCS 5125.

[7] T. A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[8] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[9] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Par. Comput.*, 24(14):1947–1980, 1998.

[10] R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. *Proc. FOCS*, pages 364–375, 1981.

[11] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Par. Distr. Comput.*, 48:96–129, 1998.

[12] J. Koster. Parasol matrices. http://www.parallab.uib.no/projects/, 1999.

[13] J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *ACM J. Exp. Alg.*, 15:1.3:1–1.3:22, 2010.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. PODC*, pages 6–6. ACM, 2009.

[15] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *Proc. PPAM*, pages 708–717, 2008. LNCS 4967.

[16] R. H. Möhring and M. Müller–Hannemann. Cardinality matching: Heuristic search for augmenting paths. Technical Report 439, Tech. Univ. Berlin, Dept. Math., 1995.

[17] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN'96*, pages 493–498, 1996. LNCS 1067.

[18] W. J. Suijlen. BSPonMPI: An implementation of the BSPlib standard on top of MPI, Version 0.3, 2010.

[19] L. G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, 1990.

[20] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.