AN EXACT PARALLEL ALGORITHM FOR THE MAXIMUM CLIQUE PROBLEM

PANOS M. PARDALOS, JONAS RAPPE, AND MAURICIO G.C. RESENDE

ABSTRACT. In this paper we present a portable exact parallel algorithm for the maximum clique problem on general graphs. Computational results with random graphs and some test graphs from applications are presented. The algorithm is parallelized using the Message Passing Interface (MPI) standard. The algorithm is based on the Carraghan-Pardalos exact algorithm (for unweighted graphs) and incorporates a variant of the greedy randomized adaptive search procedure (GRASP) for maximum independent set of Feo, Resende, and Smith (1994) to obtain good starting solutions.

1. INTRODUCTION

Let $G = (V,E)$ be an undirected weighted graph where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices in G, and $E \subseteq V \times V$ is the set of edges in G. Each vertex $v_i \in V$ is associated with a positive weight w_i. For a subset $S \subseteq V$, we define the weight of S to be $W(S) = \sum_{i \in S} w_i$ and $G(S) = (S, E \cap S \times S)$ as the subgraph induced by S. The size of the vertex set is throughout this paper denoted by n. The adjacency matrix of $G(V,E)$ is denoted $A_G = (a_{ij})$, where $a_{ij} = 1$ if (v_i, v_j) is an edge in G, i.e. $(v_i, v_j) \in E$, and $a_{ij} = 0$ if $(v_i, v_j) \notin E$. The complement graph of $G = (V,E)$ is the graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(v_i, v_j) \mid v_i, v_j \in V, i \neq j \text{ and } (v_i, v_j) \notin E\}$.

A graph $G = (V,E)$ is *complete* if and only if its vertices are pairwise adjacent, i.e. $\forall v_i, v_j \in V, (v_i, v_j) \in E$. A *clique* C is a subset of V such that the induced graph $G(C)$ is complete.

The objective of the maximum clique problem is to find a clique of maximum cardinality in a graph G. The maximum clique problem has many equivalent formulations as an integer programming problem, or as a continuous nonconvex optimization problem. The simplest one is the following edge formulation:

$$\max \sum_{i=1}^{n} w_i x_i, \tag{1}$$

$$\text{s.t. } x_i + x_j \leq 1, \ \forall \ (v_i, v_j) \in \overline{E},$$

$$x_i \in \{0,1\}, \ i = 1, \ldots, n.$$

In this formulation, if $x_i = 1$, then $v_i \in C$, and if $x_i = 0$, then $v_i \notin C$. Another equivalent formulation for the unweighted case is the following indefinite quadratic problem

$$\text{global max } f(x) = \frac{1}{2} x^T A_G x, \tag{2}$$

Date: November 1997.

Key words and phrases. Maximum clique problem, exact algorithm, parallel algorithm, GRASP, Message Passing Interface.

$$\text{s.t.} \sum_{i=1}^{n} x_i = 1,\ x_i \geq 0,\ i = 1,\ldots,n.$$

Let x^* and $\alpha = f(x^*)$ be the optimal solution and the corresponding objective value of problem (2). Then G has a maximum clique C of size $k = 1/(1-2\alpha)$. The global maximum of (2) can be attained by setting $x_i^* = \frac{1}{k}$ if $v_i \in C$, and $x_i^* = 0$ otherwise. A similar nonlinear programming formulation has been recently obtained for the weighted maximum clique problem [17].

The weighted maximum clique problem asks for the clique of maximum weight. An independent set (stable set, vertex packing) is a subset of V whose vertices are pairwise nonadjacent. The objective of the maximum independent set problem is to find a largest cardinality independent set. In the presence of weights, we seek a largest weighted independent set. A vertex cover S is a subset of V that covers all the edges of G, i.e. $\forall (v_i, v_j) \in E$ has at least one endpoint in S. In the minimum vertex cover problem, one seeks a cover of minimum cardinality. In the minimum weighted vertex cover problem one wants to find the vertex cover of minimum weight.

These problems are computationally equivalent. C is a clique in a graph G if and only if C is an independent set in the complement graph, $\bar{G} = (V, \bar{E})$, and if and only if $V \setminus C$ is a vertex cover of $\bar{G}$. Furthermore, all of these problems are known to be NP-complete [4, 14].

The maximum clique problem has been approached with exact and heuristic approximation techniques. Since the problem is NP-complete, one can expect exact solution methods to have limited performance on large dense problems. On the other hand, without an upper bound, one can never know how close a heuristic solution is to a maximum clique.

Robson [29] has developed a recursive algorithm for the maximum independent set problem with a time complexity upper bound of $O(2^{0.276n})$, where n is the number of vertices in the input graph. This exact algorithm has the best known complexity bound but no experimental evidence of its performance is known. A computationally efficient exact algorithm for the unweighted case has been proposed by Carraghan and Pardalos [8].

The main difficulty with heuristic approximation of the maximum clique problem is that a local optimum can be far from a global optimum. This difficulty is overcome by many heuristics with designs that allow escape from poor local optimal solutions. One heuristic that contains such a device is GRASP [10, **?**, **?**]. Another interesting heuristic is based on a continuous method [16, 17]. For further information on various algorithms and heuristics see [15, 27].

The remainder of this paper is organized as follows. In Section 2 we review specific applications of the maximum clique problem. The exact algorithm of Carraghan and Pardalos is review and extended to weighted graphs in Section 3. In Section 5, details of the parallel implementation of the algorithm, using the MPI standard, are presented. Experimental results are presented in Section 6 and concluding remarks are made in Section 7.

2. APPLICATIONS

The maximum clique problem has many practical applications in science and engineering. These include project selection, classification, fault tolerance, coding, computer vision, economics, information retrieval, signal transmission, and alignment of DNA with protein sequences. Test problems originating from some of these applications are available in [23].

The retrieval of similar data is an obvious application of the maximum clique problem. A graph is constructed with vertices corresponding to data items and the edges connect

vertices that are similar. A clique in such a graph is a cluster. Examples of such problems are the identification and classification of new diseases based on symptom correlation [5], computer vision [2], and biochemistry [26, 35]. In biochemistry, or more specifically, in the multiple alignment of protein sequences, the problem is to identify portions of distinct gene sequences in the DNA that are similar to a given protein. In Takefuji et al. [33], maximum independent sets in derived graphs are used to predict the structure of ribonucleic acids. More recent work on applying maximum clique algorithms for matching three-dimensional molecular structures is discussed in [13].

A major application of the maximum clique problem occurs in the area of coding theory [6, 30]. The goal here is to find the largest binary code, consisting of binary words, that can correct a certain number of errors. Each word in the code is a vector of length n. The Hamming distance between two vectors $u = (u_1, u_2, \ldots, u_n)$ and $v = (v_1, v_2, \ldots, v_n)$ is the number of components for which the two vectors differ. It is known that a code consisting of a set of binary words such that any two words have Hamming distance greater or equal to d can correct $\lfloor \frac{d-1}{2} \rfloor$ errors. Let $A(n,d)$ be the maximal number of binary words of length n and Hamming distance $\geq d$. Then $A(n,d)$ can be computed by constructing a graph consisting of 2^n vertices, corresponding to all possible code-words of length n. Two vertices, in the graph, are defined to be adjacent if their Hamming distance is at least d. The maximum clique of this graph gives the maximum number of binary vectors that can detect $\lfloor \frac{d-1}{2} \rfloor$ errors.

Another application arises in geometry. A family of hypercubes with disjoint interiors and whose union is R^n is called a tiling [32]. If the centers of the cubes form a lattice, the tiling is a lattice tiling. Minkowski conjectured that in a lattice tiling of R^n with unit n-cubes, there must exist two cubes that share an $(n-1)$-dimensional face. Minkowski's conjecture was proved by Hajós [22] in 1942. In 1930, Keller generalized the conjecture, suggesting that it holds even without the lattice assumption. Corradi and Szabo [9] proved that there is a counterexample to Keller's conjecture, if and only if, the graph Γ_n with 4^n vertices has a maximum clique of size 2^n. The graph Γ_n is defined as the graph with vertex set V of n-tuples of integers 0, 1, 2 and 3, i.e. $V_n = \{(d_1, d_2, \ldots, d_n) : d_i \in \{0,1,2,3\}, i = 1,2,\ldots,n\}$. Two vertices $u = (d_1, d_2, \ldots, d_n)$ and $v = (d'_1, d'_2, \ldots, d'_n)$ are adjacent, if and only if, the corresponding components of the n-tuples in one position have the relation 2 mod 4 and if there is another position in which the components differ. Perron [28] has shown that Keller's conjecture holds for $n \leq 6$ and Lagarias and Shor [25] have proved that it fails for $n \geq 10$. Thus, it is left to prove whether the conjecture holds for $n = 7$, 8 and 9.

Clique detection can be used as a subproblem for distributed fault diagnosis in multiprocessor systems [3]. The task is to identify a faulty processor. It is assumed that a fault-free processor in the system detects a faulty processor with some probability, while no assumptions are made on the performance of faulty processors. A major step in the algorithm is to find the maximum clique in an appropriate graph (a c-fat ring).

Determining maximum cliques is also very useful in circuit design. The problem is to create an optimal geometric layout for different chip hardwares, such as programmable logic arrays and CMOS transistors. Fairly sophisticated modeling is done to construct the graphs whose maximum cliques yield solutions to the original design problem.

3. AN EXACT ALGORITHM

A very simple and effective algorithm for the maximum clique problem has been proposed by Carraghan and Pardalos [8]. This algorithm was used as a benchmark in the Second DIMACS Implementation Challenge [24]. The algorithm initially searches the whole

graph G considering the first vertex v_1 and finds the largest clique C_1 that contains v_1. Then v_1 is not further considered since it is not possible to find a larger clique containing v_1. The algorithm next searches the graph $G - \{v_1\}$ considering v_2 and finds C_2, the largest clique in this subgraph that contains v_2. The algorithm proceeds until no clique, larger than the incumbent, can be found. The algorithm can be extended to handle weighted graphs and is highly parallelizable. This is the subject of the remainder of the paper.

3.1. **Unweighted maximum clique problem.** Initially, the algorithm orders the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ in G. The vertices are ordered so v_1 is the vertex of smallest degree in G, v_2 is the vertex of smallest degree in $G \setminus \{v_1\}$, and generally v_k is the vertex of smallest degree in $G \setminus \{v_1, v_2, \ldots, v_{k-1}\}$, for $k \leq n-2$, where $n = |V|$. It has been observed that for dense graphs the computational time is reduced if the vertex of smallest degree is considered first. The ordering is done if the density of the graph is greater or equal to 0.4. For sparse problems the algorithm is faster without any ordering.

Crucial to understanding the algorithm is the notion of *depth*. At depth 1 all the vertices are considered. The algorithm expands these vertices one at a time. Suppose that at depth d vertex $v_{di} \in V_d$ is expanded, where $V_d = \{v_{d1}, v_{d2}, \ldots, v_{di}, \ldots, v_{dm}\}$ is the set of all vertices that are considered at depth d. Next, the depth is increased by one and all vertices adjacent to the expanded vertex v_{di} and included in V_d are considered at depth $d+1$. A new vertex in V_{d+1} is now expanded at the new depth.

At depth d we have a list of vertices, $v_{1i}, v_{2j}, \ldots, v_{dk}$, such that all vertices are adjacent with each other in G, i.e. they form a clique. Thus, if every vertex is expanded as deep as possible, the maximum clique will be eventually found. To speed up the search process, the algorithm uses *pruning* to reduce the search space. The idea is to discover whether it is possible to compute a larger clique than the current best clique (*CBC*) by expanding the remaining vertices at depth d. If no larger clique can be found, the subproblems will be ignored and the algorithm will return to the previous depth.

Let d be the current depth, v_{di} the vertex that is currently expanded at step i, and let V_d be the set of vertices that are considered at depth d. Then, if

$$d + (m - i) \leq |CBC|$$

the algorithm will prune. The algorithm returns to depth $d-1$ and expands the next vertex in line at this depth. When it is possible to prune at depth 1 the maximum clique in G has been found and the algorithm terminates.

This algorithm can be further improved by initially running a heuristic to get a lower bound on the maximum clique. If it is known that the maximum clique has size $\geq \alpha$, this lower bound can be used as a pruning condition until a better clique is found. The algorithm will prune when $d + (m - i) \leq \alpha$. If α is close to the actual maximum clique, the computational time can be greatly reduced for dense graphs.

3.2. **Weighted maximum clique problem.** For the weighted case, as in the unweighted case, it is possible to improve the algorithm's pruning capabilities by ordering the vertices. In weighted graphs, all vertices are associated with non-negative weights, $w_1, w_2, \ldots, w_n$. The ordering is done so that v_1 is the vertex of largest weight, v_2 is the vertex of second largest weight, and so on, and v_n is the vertex of smallest weight. This ordering is always done regardless of the density of the graph.

If the problem is to find a weighted maximum clique in a weighted graph, the pruning condition will be different from the one in the unweighted case. Let d be the current depth

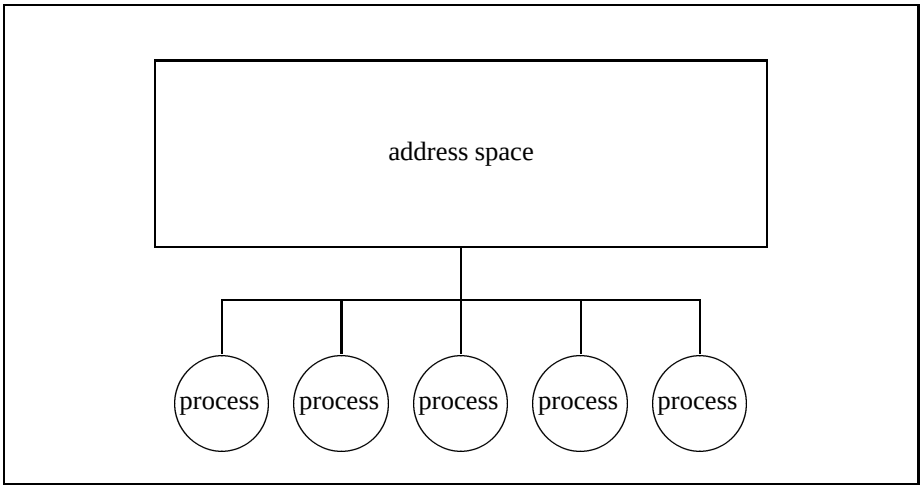

FIGURE 1. Shared memory space

and w_{di} be the weights of the vertices that are left to expand. If

$$\sum_{k=1}^{d-1} w_{ki} + \sum_{i=1}^{m} w_{di} \leq \sum_{j \in CBC} w_j$$

then the algorithm will prune, i.e. the algorithm prunes when the weight of the current clique plus the weight of the remaining nodes at the current depth d is less or equal to the weight of the current best clique *CBC*.

4. PARALLEL COMPUTING AND MPI

In parallel programming, the existence of many processors is exploited [11]. The idea is to divide the workload among several processors in order to make the program run faster. One of the largest problems in building parallel algorithms is load balancing. The performance of a parallel algorithm can be measured by its *speedup*. Let T_1 be the running time for the program on one processor and T_p be the running time for the program on p processors. Then the speedup is usually defined as T_1/T_p.

4.1. **Parallel Computational Models.**

There are several parallel computational models which can be implemented on modern parallel computers. The models form a complicated structure and differ from each other in many ways, including whether memory is physically shared or distributed, how much communication is in hardware or software, and so forth. The principal parallel computational models are data parallelism, shared memory, and message passing.

Parallelism was first made available to programmers in vector processors (data parallelism). The vector machine operates on an array of similar data items in parallel. This has been extended to include the operation of whole programs on collections of data structures (single instruction, multiple data, or SIMD). The parallelism comes entirely from the data and the program looks very much like a sequential program.

In the shared memory model, each processor has access to a single, shared address space (see Figure 1). The coordination of access by several processors to the same memory location is done by some form of locking, although this may be hidden by the programming

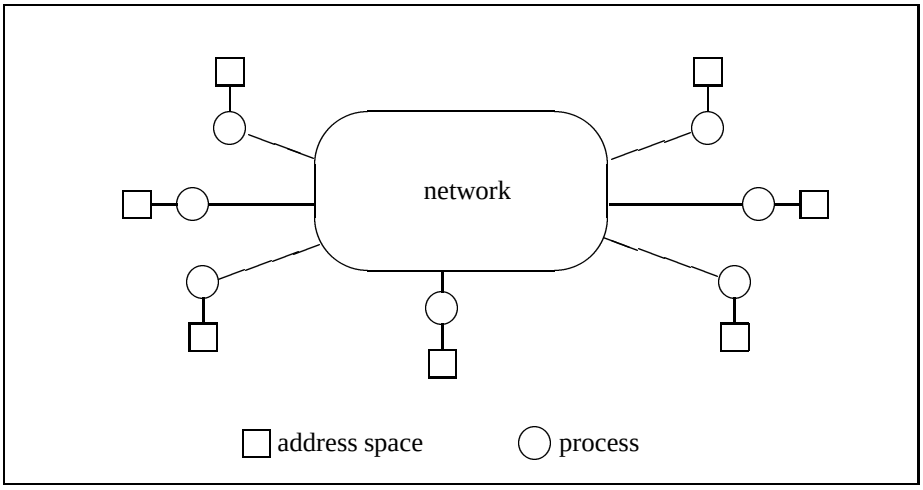

FIGURE 2. Message passing model

language. It is difficult to build "true" shared-memory machines with more than a few tens of processors. If the number of processors is large, one must allow some memory references to take more time than others. A variation of the shared memory model is to let the processors have local memory and share a part of the main memory.

The message passing model has a set of processors that have only local memory but are able to communicate with each other by sending and receiving messages. The different processors are connected by a network (see Figure 2). In the message-passing model, the sending processor and the receiving processor must perform an operation for a message to be transferred. This model becomes highly portable since it matches the hardware of most modern supercomputers as well as networks of workstations.

4.2. **MPI.** The Message Passing Interface (MPI) is a portable message passing standard for building parallel applications. The standard defines library routines and macros that can be used in C or Fortran programs.

MPI was developed in 1993–1994 by the Message Passing Interface Forum, a group of researchers representing vendors of parallel systems, industrial users, industrial and government research laboratories, and universities. More than 80 people from 40 organizations were involved in the development of the MPI standard [12]. There are several implementations of MPI that can run on distributed-memory multiprocessors and shared-memory multiprocessors, as well as on networks of workstations. These machines can be used in any combination.

The MPI standard is a large library, including more than 125 functions, which specifies the communication between a set of processes that forms a concurrent program. Since the message-passing paradigm is used, the program becomes widely portable and scalable. A complete description of the MPI can be found in [21, 31].

The standard includes point-to-point communication, collective communications, process groups, communication domains, process topologies, environmental management and inquiry, profiling interface, and bindings for Fortran and C. The standard does not specify explicit shared-memory operations, debugging facilities, explicit support for threads, support for task management, and I/O functions.

4.2.1. *Point to Point Communications.* MPI contains a set of send and receive functions, e.g. MPI_SEND and MPI_RECV, that allow communication between pairs of processes. The message from the source process contains the *type* of data, a *tag* and a *communicator*. The type of message is necessary in order to specify the correct data representation when a message is sent from one architecture to another. The tag makes it possible to choose between different messages at the receiving process. One can receive on a particular tag or choose to receive on any tag. The communicator defines the set of processes that are allowed to take part in a communication operation.

MPI_SEND and MPI_RECV are *blocking* send and receive functions. The send call blocks until the send buffer can be reclaimed, i.e. until the message is actually sent. In the same way, the receive call blocks until the receive buffer contains the message. This means that a process cannot do any work while sending or receiving data. MPI also contains *non-blocking* send and receive functions that make it possible to overlap message transmittal with computation or multiple message transmittals with one another. The non-blocking functions always contain two parts, the posting part which begins the operation and the test part which checks if the operation has completed.

Point to point communication functions have four different *modes*. The modes allow the user to choose the behavior of the send operation. In *standard* mode, the send operation can complete while the matching receive may not even have started, and the MPI does not guarantee that the data is buffered. In *buffered* mode, the user can provide a certain amount of buffering space. This must be provided by the application program. In *synchronous* mode, the completion of the send implies that the receive has at least been initiated. The last mode is the *ready* mode. In this mode the user asserts that the receive has already been called when the send call is made.

4.2.2. *Collective Communications.* In order to transmit data among all the processes specified by a communicator one can use collective communications. MPI provides several collective communication functions. MPI_BARRIER synchronizes the processes without passing any data, MPI_BCAST sends the same data from a single process to every other process, MPI_GATHER gathers data from all processes to one process, and MPI_SCATTER distributes data in a send buffer to all the other processes. Global reduction operations can be made by MPI_REDUCE. It takes data from all processes and computes the result of a reduction operation, such as sum, maximum or minimum, and then sends the result to one process. Other collective functions are combinations of these functions. Examples of these are MPI_ALLGATHER, which is a MPI_GATHER followed by a MPI_BCAST of the gathered data, and MPI_ALLTOALL, which is a set of MPI_GATHERs, where each process receives a different result from all other processes.

The collective communication functions are in many ways more restrictive than point to point functions. One restriction is that the amount of data sent from one process must exactly match the amount specified by the receiver. Another simplification is that the collective functions only exist in blocking versions.

Finally, the collective functions do not come in different modes. The mode used for collective functions is like the standard mode for point to point functions. The collective function, on a given process, is free to return as soon as it has done its part of the overall communication. This does not mean that other processes have completed or even started the operation, i.e. a collective communication may, or may not, synchronize all the calling processes. MPI_BARRIER is an exception to this.

4.2.3. *User-defined Datatypes.* All MPI communication functions take a datatype argument. This can be a primitive type like an integer or a floating point number, but it can

also be a user-defined complex type. The user-defined types are called *derived datatypes*. The derived datatypes are not types to the programming language. They are only types in the sense that MPI is aware of them and that they describe the locations of the different parameters in memory. These types are used to communicate complex data structures such as sections of arrays and combinations of different primitive datatypes.

The user defined types are constructed by type-constructing functions. The most general function is the MPI_TYPE_STRUCT, which can create a type, consisting of different primitive types that can be randomly placed in memory. The user must provide a complete description of each element of the type. There also exist other type-constructing functions. The MPI_TYPE_CONTIGUOUS builds a type whose elements are contiguous entries in an array. MPI_TYPE_VECTOR takes entries that are equally spaced in an array, and MPI_TYPE_INDEXED builds types where the elements can be arbitrary entries in an array.

4.2.4. *Communicators.* A communicator is a set of processes that can send messages to each other. At startup, MPI provides a standard communicator, MPI_COMM_WORLD. This communicator include all the processes in the program. In many cases, such as library routines and modules, it is very useful to be able to treat a subset of MPI_COMM_WORLD as a communication universe. This can be done by defining new communicators.

A communicator is, in its simplest form, composed of a *group*, which is an ordered set of processes, and a *context*, which is a system-defined tag that is attached to the group. In other words, two processes can communicate with each other if they belong to the same group and use the same context. A new communicator is created by building a new group with MPI_GROUP_INCL and then calling MPI_COMM_CREATE, which associates a context with the new group. Another function for creating communicators is MPI_COMM_SPLIT, with which it is possible to create many communicators at the same time.

Additional information can be associated with a communicator. This information is said to be *cached* with the communicator. The most important information that can be cached with a communicator is a topology. This is a structure that makes it possible to address the processes in different ways. There are two types of topologies that can be created in MPI, a *grid* topology and a *graph* topology.

4.3. **Implementations of MPI.** Recently, several implementations of MPI have appeared, both free and commercial. One free implementation is MPICH [20], which is available from Argonne National Laboratory. MPICH runs on distributed-memory machines, shared-memory machines, as well as on networks of workstations. Some other implementations are LAM [7] from the Ohio Supercomputer Center, CHIMP-MPI [1] from Edinburgh Parallel Computing Center, and Unify [34] from Mississippi State University.

MPICH was developed by William Gropp and Ewing Lusk at the same time as the work with the MPI Standard was in progress. The MPICH implementation was immediately available when the MPI Standard was released in May 1994. It can freely be downloaded from Argonne National Laboratory. For information on how to install and use MPICH, see the installation guide [18] and the user's guide [19].

5. PARALLEL EXACT ALGORITHM

We describe the parallelization of the Carraghan-Pardalos algorithm with MIP. The program has been written in Fortran 77. Since MPI has been used, the program will run on most modern parallel computers, as well as on networks of homogeneous and heterogeneous workstations.

```
Program Maximum Clique with MPI
1   include mpif.h
2   MPI_INIT(ierr)
3   MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
4   MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
5   master ← 0
6   clqblls ← block lengths of types to be included in clqtype
7   MPI_ADDRESS(maxwgt2, clqdispl(1))
8   MPI_ADDRESS(clqsiz2, clqdispl(2))
9   MPI_ADDRESS(subprb, clqdispl(3))
10  MPI_ADDRESS(best2, clqdispl(4))
11  MPI_TYPE_HINDEXED(4, clqblls, clqdispl, MPI_INTEGER, clqtype, ierr)
12  MPI_TYPE_COMMIT(clqtype, ierr)
13  wgtblls ← block lengths of types to be included in wgttype
14  MPI_ADDRESS(wgtlft, wgtdispl(1))
15  MPI_ADDRESS(maxwgt, wgtdispl(2))
16  MPI_TYPE_HINDEXED(2, wgtblls, wgtdispl, MPI_INTEGER, wgttype, ierr)
17  MPI_TYPE_COMMIT(wgttype, ierr)
18  if (master) then
        Master initiates and assigns different vertices to expand to
        the slaves.
        ...
19  else
        Slaves compute the best clique including the assigned vertex.
        ...
20  end if
21  MPI_FINALIZE(ierr)
22  end
```

FIGURE 3. Common part of parallel algorithm

5.1. **Master-Slave algorithm prototype.** The program uses the *master-slave* algorithm prototype, i.e. it uses one processor as the master process and the rest as slave processes. The master distributes the subproblems among the slaves, which do the actual computational work. As soon as a slave is done with its subproblem, it sends the result back to the master who returns to it a new subproblem. This way of building a parallel program is particularly appropriate when the slave processes do not have to communicate with each other and when the amount of work that each slave has to perform is difficult to predict. In the case of our algorithm both of these criteria hold.

The master-slave prototype will work well as long as the master process can keep up with the slave processes. If the master is communicating with one of the slaves when another slave is finished with its work, the second slave process will be idle until the master can receive its result. This means that if there are too many slave processes the benefit from sharing the work among many processors will be reduced and the speedup will decrease.

5.2. **Implementation.** The master process and the slave processes execute distinct algorithms. The different algorithms are combined into a single program. A test near the beginning separates the master code from the slave code. The slave processes are assigned different vertices to expand on. As soon as a slave process is done with a vertex, it sends the result back to the master process which immediately returns a new subproblem for the slave to work on. The pseudo code of the program is here presented in three parts, first one

common part which is executed by all the processes, then the part which is executed only by the master, and finally the part which is executed only by the slaves.

5.2.1. *Common Part.* The common part of the algorithm is presented in Figure 3. A file `mpif.h`, which defines various variables and constants that is necessary in a MPI Fortran program is included and MPI_INIT is called. This must always be the first MPI-call in every MPI program. MPI_INIT takes only one argument (`ierr`) which is an error code that is returned by every MPI subroutine. After the initiation, a call to MPI_COMM_SIZE is made (line 3). This function takes a communicator as the first argument and returns the number of processors (`nprocs`) in its second argument. In this case, the communicator is MPI_COMM_WORLD. It includes all the processors in the program and is provided by MPI at startup. Each processor determines its rank in the group associated with the communicator by calling MPI_COMM_RANK. These ranks are return in `myid`. They are consecutive integers starting at 0. Each processor will have a different number for `myid`, which will be used to separate the master process (process 0) from the slave processes.

The algorithm builds two user-defined datatypes, `wgttype` and `clqtype`. These types are used to send data between the master and the slaves. The data is gathered in the complex types so there only has to be one send and one receive call at every data transfer. `clqtype` includes `maxwgt2`, `clqsiz2` and `subprb` which are integers, and `best2` which is an array of integers. The length of these four blocks of the new datatype is put in the array `clqblls`. In this case the three first lengths are 1 and the last one is the maximum size of the array. The address in memory for the different blocks is returned into an array `clqdispl` by consecutive calls to MPI_ADDRESS. When the addresses are known, the datatype can be created by calling MPI_TYPE_HINDEXED which is a type-constructing function that builds derived datatypes with arbitrary entries in memory but only takes one type of primitive datatype. MPI_TYPE_HINDEXED takes, as arguments, the number of blocks, the lengths of the different blocks, the blocks displacement in memory, the MPI primitive datatype and the name of the new type. Finally, the type has to be committed by calling MPI_TYPE_COMMIT and giving the new type as an argument. `clqtype` is constructed in lines 6–12 and then `wgttype` is constructed in the same way in lines 13–17.

In line 18, `myid` is compared with `master` in order to let the number 0 processor execute one code and the other processors another code. The last thing in the program is to call MPI_FINALIZE. This function must be called by every processor so the MPI "environment" can be terminated. No MPI call can be made after the call to MPI_FINALIZE.

5.2.2. *Master Part.* The master algorithm (Figure 4) begins by reading the graph from a file by calling the subroutine `readgraph`. The input graph can consist of either weighted or unweighted vertices. If it is an unweighted graph, the master calls the GRASP for the unweighted maximum clique problem, to get a lower bound for the maximum clique (lines 2–4). In the next step, the vertices in the graph are ordered. If the graph is weighted, then the vertices are ordered in decreasing weight order. If the graph is unweighted and dense, then the vertices are ordered by increasing degree. The ordering is done by calling `ordmatwgt` and `ordmatdeg` respectively (lines 5–11). In line 12, the number of vertices, the number of edges, and the number of processors used, are written to the output file. Next, a check is made to ensure that the size of the vertex set is not larger than the number of processes (lines 13–15). The total weight of the graph is put into `wgtlft`(line 16). For an unweighted graph this will be equal to the number of vertices.

The number of vertices n and the graph which has been stored in `matrix` is broadcast to all the slaves by using the collective communication function MPI_BCAST. The arguments to MPI_BCAST are the variable to be sent, the size of the variable, the type of the variable,

TABLE 1. Computational results on weighted random graphs

Vertices	Edges	Graph density (%)	Weight of clique	Size of clique	CPU-times 2 proc (sec)	CPU-times 4 proc (sec)	Speedup
100	3,972	0.80	125	18	19.04	10.83	1.75
100	4,464	0.90	196	30	517.08	259.13	2.00
200	11,975	0.60	98	12	16.41	8.95	1.83
200	13,957	0.70	126	17	143.37	55.53	2.58
200	15,920	0.80	165	22	5382.35	2050.08	2.63
300	22,387	0.50	89	11	23.28	12.18	1.91
300	27,005	0.60	109	12	219.59	82.01	2.68
300	31,532	0.70	141	17	5204.66	2108.67	2.47
400	31,701	0.40	75	9	17.55	10.85	1.62
400	39,698	0.50	93	11	106.69	48.54	2.20
400	47,973	0.60	119	14	1472.82	557.21	2.64
500	49,675	0.40	83	9	46.24	21.79	2.12
500	62,130	0.50	102	11	397.16	160.31	2.48
500	74,983	0.60	129	14	7601.54	2903.21	2.62

TABLE 2. Computational results on unweighted random graphs

Vertices	Edges	Graph density (%)	Size of clique	Size of GRASP clique	CPU-times 2 proc (sec)	CPU-times 4 proc (sec)	Speedup
100	3,972	0.80	20	19	35.78	16.48	2.17
100	4,230	0.85	24	24	183.24	70.13	2.61
100	4,464	0.90	30	29	1731.35	666.27	2.60
200	11,975	0.60	14	13	64.16	24.42	2.63
200	13,957	0.70	18	17	929.04	353.01	2.63
300	17,918	0.40	9	9	14.79	8.14	1.82
300	22,387	0.50	12	11	66.72	28.44	2.35
300	27,005	0.60	15	15	1033.63	388.80	2.66
400	31,701	0.40	10	9	48.56	21.00	2.31
400	39,698	0.50	13	11	384.79	151.10	2.55
400	47,973	0.60	16	15	9213.88	3466.43	2.66
500	37,335	0.30	8	8	18.56	10.13	1.83
500	49,675	0.40	11	9	121.07	51.12	2.37
500	62,130	0.50	13	13	1452.71	584.03	2.48

the rank of the source processor, the communicator in which both the receiving and the sending processors must be included, and an error code. This call to MPI_BCAST must be matched by an identical call in the slave processes if the transfer is to be completed.

In lines 19–22, `wgttype` is sent to one slave process at a time. MPI_SEND has as arguments an address, the number of elements to be transferred, the type of the elements, the rank of the receiving processor, a tag, a communicator, and an error code. The address MPI_BOTTOM is a reference address to `wgttype`. The communicator must contain both the source process and the destination process. The call is tagged with `node` to let the slave processes know which vertex to expand. The call to MPI_SEND at the master must

TABLE 3. Computational results on Hamming graphs

n	d	Vertices	Edges	Graph density (%)	GRASP clique	Max clique	CPU time 2 proc (sec)	CPU time 4 proc (sec)	Speedup
6	2	64	1824	0.90	32	32	6.79	5.34	1.27
8	4	256	20864	0.64	16	16	421.12	166.85	2.52

be matched by a call to MPI_RECV on the receiving slave process. Between the calls to MPI_SEND, `wgtlft` is decreased by the weight of `node`.

The main loop of the algorithm is executed on lines 23–31. While the weight of the maximum clique (`maxwgt`) is greater than the weight of the remaining vertices in the graph (`wgtlft`), subproblems are assigned to the slave processes. The master process receives a clique from a slave process when MPI_RECV is called. MPI_RECV takes the same arguments as MPI_SEND, and in addition, has a status argument (`status`), which is an array that contains the source and the tag of the received message. If the received clique is larger than the current best clique, then the maximum clique is updated. After this check, a new subproblem is sent to the free slave process, `wgtlft` is decreased by the weight of the sent vertex, and `node` is increased by one.

When `maxwgt` is less or equal to `wgtlft` it is no longer necessary to expand the graph any further. The master waits for the remaining slaves to send the result of the vertices they are currently expanding. If the received clique is larger than the current best clique, then the maximum clique is updated. When a clique is received, the master sends another message to the slave so the slave process can be terminated (lines 32–38). In lines 39–40 the user defined datatypes are freed and on line 41 the master prints the result on an output file.

5.2.3. *Slave Part.* The computational work is done by the slave processes. Their part of the algorithm is described in Figure 5. The slaves receive n and `matrix` from the master by calling MPI_BCAST (lines 1–2). These calls are matched with corresponding calls on the master process. In lines 3–7 the slave receives a subproblem, computes the maximum clique, and returns the result as long as `maxwgt` is less or equal to `wgtlft`. The subproblem is received by calling MPI_RECV, which is tagged with `node` in order to inform the slave process on which vertex to expand. This call is also matched with a corresponding call on the master process. Next, `expand`, the function that computes the maximum clique including `node`, is called. The result is sent back to the master by calling MPI_SEND. When `maxwgt` is greater than `wgtlft` the algorithm has found the maximum clique and terminates.

6. COMPUTATIONAL RESULTS

In this section, preliminary computational results are presented using the parallelized algorithm. The algorithm was implemented in Fortran 77 and ran on a network of Sun 4 workstations. The algorithm has been tested on both weighted and unweighted graphs and the size of the problems vary from 64 vertices with 1,824 edges to 500 vertices with 74,983 edges.

We used in the experiments two and four processors. Let T_2 be the CPU time for two processors and T_4 be the CPU time for four processors. The speedup used in the tables is defined by T_2/T_4. If the speedup were perfect, the ratio would be very close to 3, since

TABLE 4. Computational results on Keller 4 graph

n	Vertices	Edges	Graph density (%)	Size from GRASP	Size of clique	CPU time 2 proc (sec)	CPU time 4 proc (sec)	Speedup
4	171	9435	0.65	11	11	80.289	33.362	2.41

when two processors are used all the computational work is done by a single slave process, while with four processors the computations are done by three slave processes.

Table 1 presents results with weighted random graphs and Table 2 with unweighted random graphs. Both the weighted and the unweighted random graphs were generated using the standard IBM random number generator GGUBFS. For the weighted graphs (Table 1), the speedup ranges from 1.62 to 2.68, while for larger problems it is around 2.5. The algorithm shows poor performance for small test problems because a large part of the CPU time is used to initialize MPI, building datatypes, and so on. For the unweighted problems, the speedup is between 1.82 and 2.66, but again it improves for larger problems. By comparing the two tables, one can see that weighted problems are more easily solved than unweighted problems of the same size. This is due to the fact that the pruning criterion used is a function of the weights of the vertices, i.e. it is possible to prune earlier in a weighted problem than it is in an unweighted problem.

In Table 3, the computational results on Hamming graphs are presented. The first column is the size (n) of the binary vector and the second is the Hamming distance (d) between any two vectors. The speedup is 1.27 and 2.52, respectively, and again the speedup is around 2.5 for the larger problem.

In Table 4, a computation with Keller 4 is summarized. Due to symmetry in the graph, the number of nodes is smaller than 4^4. The maximum clique will still be the same as for the original Keller graph. The speedup for Keller 4 with four processors is 2.41.

It is interesting to note that if more processors are used the algorithm has to solve more subproblems. This is due to the fact that old pruning conditions have to be sent to the slaves since the result from the previous node is not yet available.

7. CONCLUDING REMARKS

In this paper, we present an exact parallel algorithm for the maximum clique problem. Since the MPI is used to parallelize the algorithm, the code can run on many advanced parallel machines, as well as on networks of workstations.

The algorithm has been tested on a variety of test problems and it has been observed that its performance improves, as the size (number of vertices and density) of the problem increases.

The source code is available from the authors.

REFERENCES

[1] R. Alasdair, A. Bruce, J.G. Mills, and A.G. Smith. CHIMP/MPI user guide. Technical Report EPCC-KTP-CHIMP-V@-USER 1.2, Edinburgh Parallel Computing Center, 1994.

[2] D.H. Ballard and M. Brown. *Computer Vision*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[3] P. Berman and A. Pelc. Distributed fault diagnosis for multiprocessor systems. In *Proc. of the 20th Annual Intern. Symp. on Fault-Tolerant Computing*, pages 340–346, Newcastle, UK, 1990.

[4] P. Berman and G. Schnitger. On the complexity of approximating the independent set problem. *Lecture Notes in Computer Science*, 349:256–267, 1989.

[5] R.E. Bonner. On some clustering techniques. *IBM J. of Research and Development*, 8:22–32, 1964.
[6] A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith. A new table of constant weight codes. *J. IEEE Trans. Information Theory*, 36:1334–1380, 1990.
[7] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
[8] R. Carraghan and P.M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.
[9] K. Corradi and S. Szabo. A combinatorial approach for keller's conjecture. *Periodica Mathematica Hungarica*, pages 95–100, 1990.
[10] T.A. Feo and M.G. Resende. Greedy randomized adaptive search procedures. *J. of Global Optimization*, 6:109–133, 1995.
[11] A. Ferreira and P.M. Pardalos, editors. *Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques*, volume 1054 of *Lecture notes in computer science*. Springer-Verlag, 1996.
[12] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International J. of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994.
[13] E.J. Gardiner, P.J. Artymiuk, and P. Willett. Clique-detection algorithms for matching three-dimensional molecular structures. *J. of Molecular Graphics and Modelling*, 1997. To appear.
[14] M. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, San Fransisco, 1979.
[15] L.E. Gibbons. *Algorithms for the Maximum Clique Problem*. PhD thesis, University of Florida, 1994.
[16] L.E. Gibbons, D. Hearn, and P.M. Pardalos. A continuous based heuristic for the maximum clique problem. In *Clique, Graph Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series on Discrete Mathematics and Theoretical COmputer Science*, pages 103–124. American Mathematical Society, 1996.
[17] L.E Gibbons, D. Hearn, P.M. Pardalos, and M.V. Ramana. A continuous characterization of the maximum clique problem. *Math. of Oper. Res.*, 22:754–768, 1997.
[18] W. Gropp and E. Lusk. Installation guide to `mpich`, a portable implementation of MPI. Technical Report ANL-96/5, Argonne National Laboratory, 1994.
[19] W. Gropp and E. Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/6, Argonne National Laboratory, 1994.
[20] W. Gropp and E. Lusk. A high-performance, portable implementation of the mpi message passing interface standard. Technical report, Argonne National Laboratory, 1996.
[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.
[22] G. Hajós. Sur la factorisation des abeliens. *Casopis*, pages 189–196, 1950.
[23] J. Hasselberg, P. M. Pardalos, and G. Vairaktarakis. Test case generators and computational results for the maximum clique problem. *J. of Global Optimization*, 3:463–482, 1993.
[24] D.S. Johnson and M.A. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series on Discrete Mathematics and Theoretical COmputer Science*. American Mathematical Society, 1996.
[25] J. C. Lagarias and P. W. Shor. Keller's cube-tiling conjecture is false in high dimensions. *Bulletin American Mathematical Society*, 27:279–283, 1992.
[26] W. Miller. Building multiple alignments from pairwise alignments. *Computer Applications in the Biosciences*, 1992.
[27] P. M. Pardalos and J. Xue. The maximum clique problem. *J. of Global Optimization*, 4:301–328, 1994.
[28] O. Perron. über lückenlose ausfullung des n-dimensionalen raumes durch kongruente würfel. *Math. Z.*, 46:1–26, 161–180, 1940.
[29] J.M. Robson. Algorithms for maximum independent sets. *J. of Algorithms*, 7:425–440, 1986.
[30] N. J. A. Sloane. Unsolved problems in graph theory arising from the study of codes. *Graph Theory Notes of New York XVIII*, pages 11–20, 1989.
[31] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Donagarra. *MPI: The Complete Reference*. The MIT Press, 1996.
[32] S. K. Stein and S. Szabó. *Algebra and Tiling, Homomorphisms in the service of geometry.* American Mathematical Society, 1994.
[33] Y. Takefuji, K. Lee L Chen, and J. Huffman. Parallel algorithms for finding a near-maximum independent set of a circle graph. *IEEE Transactions on Neural Networks*, 1(3), 1990.
[34] P.L. Vaughan, A. Skjellum, D.S. Reese, and F.C. Cheng. Migrating from pvm to mpi, part I: The unify system. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 188–495, Maclean,

Virginia, 1995. IEEE Computer Society Technical Committee on Computer Architecture, IEEE Computer Society Press.

[35] M. Vingron and P.A. Pevzner. Motif recognition and alignment for many sequences by comparison of dot matrices. *J. of Molecular Biology*, 218:33–43, 1991.

CENTER FOR APPLIED OPTIMIZATION, DEPARTMENT OF INDUSTRIAL AND SYSTEMS ENGINEERING, UNIVERSITY OF FLORIDA, GAINESVILLE, FL 32611 USA.
E-mail address: pardalos@ufl.edu

DEPARTMENT OF OPTIMIZATION AND SYSTEMS THEORY, ROYAL INSTITUTE OF TECHNOLOGY (KTH), STOCKHOLM, SWEDEN.
E-mail address: t93_raj@t.kth.se

INFORMATION SCIENCES RESEARCH, AT&T LABS RESEARCH, FLORHAM PARK, NJ 07932 USA.
E-mail address: mgcr@research.att.com

```
1   call readgraph
2   if (unweighted) then
3         call GRASP
4   end if
5   if (weighted) then
6         call ordmatwgt
7   else
8         if (dense) then
9               call ordmatdeg
10        end if
11  end if
12  call initout
13  if (nprocs > n +1) then
14        nprocs = n +1
15  end if
16  wgtlft ← weight of the input graph
17  MPI_BCAST(n, 1, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
18  MPI_BCAST(matrix, maxn·maxn, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
19  for node = 1 to nprocs – 1 do
20        MPI_SEND(MPI_BOTTOM, 1, wgttype, node, node, MPI_COMM_WORLD, ierr)
21        wgtlft ← wgtlft – weight of node
22  end for
23  while (maxwgt<wgtlft) do
24        MPI_RECV(MPI_BOTTOM, 1, clqtype, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, status, ierr)
25        if (bestclique) then
26              update maximum clique
27        end if
28        MPI_SEND(MPI_BOTTOM, 1, wgttype, status(MPI_SOURCE), node,
                MPI_COMM_WORLD, ierr)
29        wgtlft ← wgtlft – weight of node
30        node ← node + 1
31  end while
32  for j=1 to nprocs – 2 do
33        MPI_RECV(MPI_BOTTOM, 1, clqtype, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, status, ierr)
34        if (bestclique) then
35              update maximum clique
36        end if
37        MPI_SEND(MPI_BOTTOM, 1, wgttype, status(MPI_SOURCE), node,
                MPI_COMM_WORLD, ierr)
38  end for
39  MPI_TYPE_FREE(clqtype, ierr)
40  MPI_TYPE_FREE(wgttype, ierr)
41  call solout
```

FIGURE 4. Master part of parallel algorithm

```
1   MPI_BCAST(n, 1, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
2   MPI_BCAST(matrix, maxn·maxn, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)
3   while (maxwgt < wgtlft) do
4       MPI_RECV(MPI_BOTTOM, 1, wgttype, master, MPI_ANY_TAG,
                MPI_COMM_WORLD, status, ierr)
5       call expand
6       MPI_SEND(MPI_BOTTOM, 1, clqtype, master, status(MPI_TAG),
                MPI_COMM_WORLD, ierr)
7   end while
```

FIGURE 5. Slave part of parallel algorithm