Solving the Maximum Clique Problem Using PUBB

Yuji Shinano[†], Tetsuya Fujie[‡], Yoshiko Ikebe[†] and Ryuichi Hirabayashi[†]

[†] Department of Management Science, Science University of Tokyo
1-3, Kagurazaka, Shinjuku-ku, Tokyo 162, Japan
shinano@ms.kagu.sut.ac.jp

[‡] Department of Mathematical and Computing Sciences, Tokyo Institute of Technology
2-12-1, Oh-okayama, Meguro-ku, Tokyo 152, Japan

Abstract

Given an (undirected) graph $G = (V, E)$, a clique of G is a subset of vertices in which every pair is connected by an edge. The problem of finding a clique of maximum size is a classical NP–hard problem, and many algorithms, both heuristic and exact, have been proposed. While the philosophy behind the heuristic algorithms varies greatly, almost all of the exact algorithms are designed in the branch–and–bound framework. As is well known, branch–and–bound is well suited to parallelization, and PUBB is a software utility which implements a generic version of it. In this paper, we show effectiveness of parallelization of branch–and–bound for the maximum clique problem. Especially, by using PUBB with good heuristics and branching techniques, we were able to solve five previously unsolved DIMACS benchmark problems to optimality.

1 Introduction

Let $G = (V, E)$ be an (undirected) graph, where V is the set of vertices and E is the set of edges (unordered pairs of distinct vertices). Two vertices u and v are adjacent if $(u, v) \in E$. The complement of G is the (undirected) graph $\overline{G} = (V, \overline{E})$, where $\overline{E}$ is the set of unordered pairs of distinct vertices which are not members of E. A *clique* C is a subset of V such that any two distinct vertices of C are adjacent. A *stable set* S is a subset of V such that no two distinct vertices of S are adjacent. A *maximum clique* (resp. *maximum stable set*) is a clique (resp. stable set) of G whose cardinality is maximum. It is clear that finding a maximum clique of G is equivalent to finding a maximum stable set of $\overline{G}$. The maximum clique problem is NP-hard for arbitrary graphs. This problem has been very well studied, for a recent survey see [8]. Since the problem is NP-hard, the standard approach to obtain an exact solution is the *branch-and-bound algorithm* which implicitly enumerates the entire solution space. Though the branch-and-bound algorithm performs intelligent enumeration using various results on the problem obtained so far, it is so hard to obtain an optimal solution of large scale instances within a reasonable amount of time.

Parallelization of the algorithm enables us to improve the most promising algorithm which exploits the latest research results. Since the branch-and-bound algorithm has a general framework which is problem-independent, its parallelization can be implemented as a generalized software tool. Once such a kind of tool is provided, use of the tool should lead to improvement of the latest research results. Several such tools have been already developed [2, 4, 9, 10]. In the use of some tool, even the previously unsolved challenging problem instances of the QAP (Quadratic Assignment Problem) have been solved[4].

PUBB(*Parallelization Utility for Branch-and-Bound algorithms*) [9] is one of the generalized software tools. PUBB provides a skeleton of a parallel branch-and-bound algorithm and has several interfaces for user defined routines. For the development of applications, PUBB also provides a skeleton of a sequential branch-and-bound algorithm, and the interfaces for user defined routines are completely the same as in the parallel one. Therefore, by using the sequential skeleton in the development of some applications, user defined routines can be ported to parallel platform without any modification.

PUBB has shown good performance in solving the Traveling Salesman Problem(TSP) in terms of acceleration of run time to get an optimal solution (*parallel run time* is measured as elapsed time from the moment that a parallel computation starts to the moment the last processor finishes execution). However, the size of the solved instances were too small to compare to the latest research results of the TSP, because the bounding procedure used in the computation was a classical one.

This paper demonstrates the success in combining the latest results for the maximum clique problem with PUBB. Several improvements of PUBB give a very good performance, even if the bounding procedure of the problem is much faster than the transmission time of communication link used. Five previously unsolved instances from DIMACS benchmarks were solved using 51 networked workstations.

In Section 2 we give a branch-and-bound algorithm for the maximum clique problem. We implement the algorithm and tune parameters to obtain better performance in sequential. In Section 3 we port the user routines to parallel platform. In the parallel platform, we tune parameters for PUBB and make some modifications to obtain better performance in parallel. In Section 4 we present solutions of the previously unsolved instances. Finally, in Section 5, we state concluding remarks.

2 Branch-and-Bound Algorithms for the Maximum Clique Problem

In this section, we provide a framework of the branch-and-bound algorithm. In the rest of the paper, $V(G)$ denotes the set of vertices of a graph G. For $v \in V(G)$, $N(v)$ consists of the vertices of G which is adjacent to v. A member of $N(v)$ is called a neighbor of v.

We denote a subproblem of the search tree by $(G', I', \overline{z}_{G'})$, where G' is a subgraph of G and I' is a clique of G which satisfies $V(G'), I' \subseteq V$, $V(G') \cap I' = \emptyset$ and $V(G') \subseteq \cap_{v \in I'} N(v)$, and $\overline{z}_{G'}$ is an upper bound of the maximum clique size of G'. I' corresponds to the vertices which are fixed-in. The subproblem asks for a maximum clique in G'. The (original) maximum clique problem corresponds to $(G, \emptyset, \infty)$.

The branch-and-bound algorithm described below contains four crucial components. They are treated in the following subsections.

algorithm MAX_CLIQUE(G**)**;
  input: graph G;
begin
  Apply **FIND_INITIAL_SOLUTION(**G**)** to obtain a clique C^* of G;
  $z^* := |C^*|$; $\mathcal{L} := \{(G, \emptyset, \infty)\}$;
  while $\mathcal{L} \neq \emptyset$ **do begin**
    Apply **CHOOSE_SUBPROBLEM(**$\mathcal{L}$**)** to obtain a subproblem $(G', I', \overline{z}_{G'}) \in \mathcal{L}$;
    $\mathcal{L} := \mathcal{L} - \{(G', I', \overline{z}_{G'})\}$;
    if $\overline{z}_{G'} + |I'| > z^*$ **then begin**
      Apply **UPPER_BOUNDING(**G', z^***)** to obtain a coloring $\mathcal{S} = \{S_1, \ldots, S_K\}$ and an upper bound $\overline{z}$ of G';
      if $\overline{z} + |I'| > z^*$ **then**
        Apply **BRANCHING(**$G', I', \mathcal{S}, C^*, z^*, \mathcal{L}$**)** to update $\mathcal{L}$, C^* and z^*;
    end;
  end;
  return(C^***)**;
end.

2.1 Finding an Initial Solution

Many heuristic algorithms have been developed to find a large clique efficiently (see [8]). A large number of these algorithms are promising, that is, they produce an optimal solution in most problem instances. The algorithm STABJOIN, originally developed for the maximum stable set problem by Ikebe and Tamura [5], is one of the promising algorithms. STABJOIN can be applied to the maximum clique problem by complementing the graph. In our computational tests, STABJOIN produced an optimal solution or a nearly optimal solution in which the difference between the maximum clique size is one or two in most problem instances. The outline of STABJOIN, which we used to find an initial solution, can be described as follows.

algorithm FIND_INITIAL_SOLUTION(G**)**;
  input: graph G;
begin
  Find a clique C^1 in a greedy fashion;
  for $t = 1$ **to** T **do begin**
    Find a clique C^2 in $V - C^1$;
    Find a maximum clique C in $C^1 \cup C^2$ (This can be done efficiently[5]);
    $C^1 := C$;
  end;
  return(C^1**)**;
end.

Throughout the paper, T is set to 1000.

2.2 Choosing a Subproblem

There are many tree search strategies (or selection rules) to choose a subproblem from the subproblem pool $\mathcal{L}$. PUBB supports several of these strategies. Among them, the Depth First search will be mainly adopted since **BRANCHING** generates many new subproblems in general. In parallelization, some modifications will be given.

2.3 Upper Bounding Procedures

In this subsection, we describe upper bounding procedures based on the coloring and the fractional coloring of G.

Let $\mathcal{S} = \{S_1, \ldots, S_K\}$ be a set of stable sets which form a partition of $V(G)$. $\mathcal{S}$ is called a coloring (or a K coloring) of G. It is well known and easily verified that the maximum clique size is at most K, that is, a coloring size provides an upper bound of the maximum clique size. Since finding a minimum coloring for an arbitrary graph is known to be NP-hard, heuristic algorithms are often designed for the coloring bound. Among them, the greedy coloring heuristic **CH** and an algorithm of Brelaz[3], **CH_B** are implemented and tested. They can be described as follows.

algorithm CH(G**)**;
  input: graph G with vertex set $\{v_1, \ldots, v_n\}$;
begin
  $\mathcal{S} := \emptyset$;
  for $i := 1$ **to** n **do begin**
    if $v_i \notin \bigcup_{S \in \mathcal{S}} S$ **then begin**
      $S' := \{v_i\}$;
      for $j := i + 1$ **to** n **do**
        if $v_i \notin \bigcup_{S \in \mathcal{S}} S$ and $S' \cup \{v_j\}$ is a stable set **then** $S' := S' \cup \{v_j\}$;
      $\mathcal{S} := \mathcal{S} \cup \{S'\}$;
    end;
  end;
  return($\mathcal{S}$**)**;
end.

algorithm CH_B(G**)**;
  input: graph G with vertex set $\{v_1, \ldots, v_n\}$;
begin
  $W := V$;
  while $W \neq \emptyset$ **do begin**
    $U := \{v \in W \mid N(v)$ contains a maximum number of colors$\}$;
    Let v^* be a vertex of U whose number of colored neighbors is maximum;

Assign a positive integer (color) $c(v^*)$;
$W := W - \{v^*\}$;
end;
$K := \max\{c(v) \mid v \in V\}$;
for $k := 1$ **to** K **do**
$S_k := \{v \in V \mid c(v) = k\}$;
return($\mathcal{S} = \{S_1, \ldots, S_K\}$);
end.

Recently, sophisticated bounding procedures have been given in [1, 7]. They are based on a fractional coloring bound (see [1]), which is equal to or tighter than the coloring bound. Since finding a minimum fractional coloring bound is also NP-hard, heuristic algorithms are proposed in [1, 7]. Our implementation is a mixture of the algorithms in [1, 7]. In the following, T is a positive integer. (See [1, 7] for the validity and theoretical background of **FCH**.)

algorithm FCH(G**,** $\mathcal{S}$**)**;
input: graph G with vertex set $\{v_1, \ldots, v_n\}$;
coloring $\mathcal{S}$ of G;
begin
$\overline{z} := |\mathcal{S}|$;
for $t := 2$ **to** T **do begin**
for $i := 1$ **to** n **do begin**
if $^{\exists}S \in \mathcal{S}$ s.t. $S \cup \{v_i\}$ is stable **then begin**
$S' := S \cup \{v_i\}$; $\mathcal{S} := \mathcal{S} \cup \{S'\} - \{S\}$;
end;
else begin
$S' := \{v_i\}$; $\mathcal{S} := \mathcal{S} \cup \{S'\}$;
end;
end;
if $|\mathcal{S}|/t \geq \overline{z}$ **then return**($\lfloor \overline{z} \rfloor$); **else** $\overline{z} := |\mathcal{S}|/t$;
end;
return($\lfloor \overline{z} \rfloor$);
end.

Since, as already stated, the fractional coloring bound is tighter than the coloring bound, the number of subproblems will decrease when the branch-and-bound algorithm uses the fractional coloring bound. On the other hand, the time needed to calculate the upper bound will increase, so there is a trade-off relation between the use of the coloring bound and that of the fractional coloring bound. Therefore, we implemented the following upper bounding procedure proposed in [1] in order to see the effect of the upper bound procedures. Here γ is a real number between 0 and 1. It is easy to see that if $\gamma = 0$, then the fractional coloring heuristic (**FCH**) is always executed, and if $\gamma = 1$ then only the coloring heuristic (**CH**) is executed.

The determination of γ is described in Section 2.5.

algorithm UPPER_BOUNDING(G**,** z^***)**;
input: graph G;
incumbent value z^*;
begin
Apply **CH(**G**)** or **CH_B(**G**)** to obtain a coloring $\mathcal{S} = \{S_1, \ldots, S_{\overline{z}}\}$;
if $\overline{z} + |I| > z^*$ and $\gamma \cdot (\overline{z} + |I|) \leq z^*$ **then**
apply **FCH(**G**)** to obtain a new upper bound $\overline{z}$;
return($\mathcal{S}, \overline{z}$)
end.

2.4 Branching

The following branching method is due to [7].

algorithm BRANCHING($G, I, \mathcal{S}, C^*, z^*, \mathcal{L}$**)**;
input: graph G with vertex set $\{v_1, \ldots, v_n\}$;
set I of vertices, where $\{v_1, \ldots, v_n\} \cap I = \emptyset$;
coloring $\mathcal{S}$ of G;
incumbent clique C^*;
incumbent clique size $z^*(= |C^*|)$;
set $\mathcal{L}$ of subproblems;
begin
$S := S_1 \cup \ldots \cup S_{z^* - |I|}$; $\{v_1, \ldots, v_p\} := V(G) - S$;
for $i := 1$ **to** p **do begin**
$I' := I \cup \{v\}$; $G' := G[N(v_i) \setminus \{v_{i+1}, \ldots, v_p\}]$;
if $|I'| > z^*$ **then begin**
$C^* := I'$; $z^* := |C^*|$;
end;
if $|V(G')| + |I'| > z^*$ **then** $\mathcal{L} := \mathcal{L} \cup \{(G', I')\}$;
end;
return($\mathcal{L}$, C^*, z^*);
end.

2.5 Implementation Details

We implemented the components described in this section and tested random problem instances by a sequential branch-and-bound algorithm.

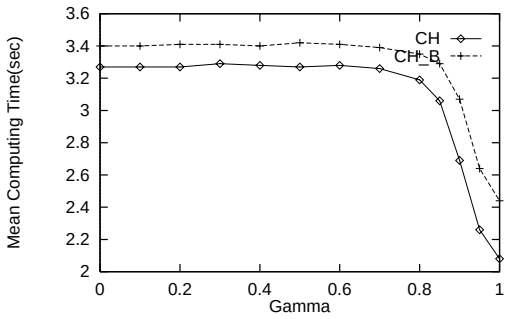

Figure 1. Mean Computing Time in the Sequential Algorithm

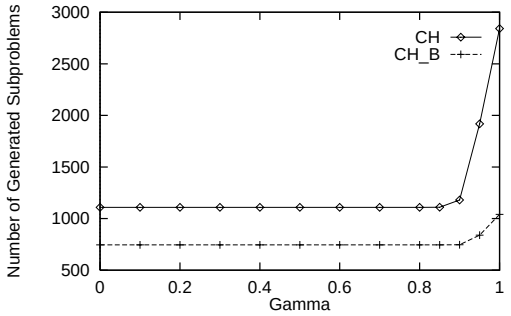

Figure 2. Number of Subproblems in the Sequential Algorithm

We generated ten random problem instances with 100 vertices and edge density 90%. Note that, as observed in [1, 7], high density (such as 80% or 90%) problems are hard to solve even if the number of vertices is only 200. Experiments in this subsection were run on a SUN Ultra compatible with UltraSPARC at 146MHz and 160M bytes memory.

Figures 1 and 2 display the effect of the coloring heuristics **CH** and **CH_B** and the real γ, by the mean computing time and the mean number of generated subproblems, respectively. In **FCH**, T is set to 4.

Figures 1 shows that, in our implementation, using the coloring heuristic only ($\gamma = 1.0$) is better than using the fractional coloring heuristic ($\gamma < 1.0$), though the number of generated subproblems is the largest when $\gamma = 1.0$ (see Figure 2). This is because, compared with the reduction in generated subproblems, the fractional coloring heuristic consumes much more time and it does not contribute to reduce the overall computing time. Therefore, in the rest of this paper, we set γ to 1.0, and thus ignore T. For the same reason, we use $\mathbf{CH}$ as the coloring heuristic. With the help of a promising heuristic for finding a clique, the most simple upper bounding procedure works in our implementation. Note that a good heuristic for finding a clique is significant to reduce not only the number of **BRANCHING** calls but also the newly generated subproblems when **BRANCHING** is executed.

3 Parallelization of the Algorithm Using PUBB

In this section, we show the behavior of the parallelized algorithm using PUBB and show how its performance is improved. In the computational experiments appearing in the rest of this paper, the workstations operated on were an IBM RS/6000 Model 25T with PowerPC601 at 66MHz and 64M bytes memory and they were connected with the Ethernet. This environment is the same as our previous work in [9].

Table 1. Computational results in the sequential algorithm(# of vertices = 200, density = 90%)

Prob. No.	Comp. Time (sec)	Evaluation Time of a Subproblem: Mean(S.D.) (sec)	# of Generated Subproblems	# of Evaluated Subproblems	# of Improved Solutions
1	24274.93	0.0012 (0.0033)	17076112	17076112	1
2	21910.94	0.0013 (0.0034)	14635886	14635886	2
3	31418.37	0.0012 (0.0033)	22242646	22242646	0
4	12842.94	0.0013 (0.0034)	8360920	8360920	0
5	23307.84	0.0012 (0.0033)	16495272	16495272	0
6	9959.94	0.0011 (0.0031)	7862052	7862052	0
7	25392.77	0.0013 (0.0034)	17019196	17019196	0
8	23601.35	0.0013 (0.0033)	16161238	16161238	0
9	19017.74	0.0012 (0.0033)	13572984	13572982	1
10	17147.96	0.0013 (0.0033)	11570975	11570975	0

For all performance evaluations in the following subsections, ten random problem instances with 200 vertices and edge density of 90% are used. Table 1 shows the computing time of these problem instances in the sequential branch-and-bound algorithm.

3.1 PUBB

PUBB is composed of three kinds of tasks and runs on PVM (Parallel Virtual Machine). Tasks of PUBB consists of **Problem Manager(PM)**, **Load Balancer(LB)** and **Solver**. There is one **PM** in the system, while there are many **LB**s and **Solver**s in it. There is a one to one correspondence between **LB** and **Solver**, and corresponding pairs run on the same workstation (see Figure 3). PUBB has three *running modes* – Master-Slave(MS) mode, Fully-Distributed(FD) mode and Master-Slave to Fully-Distributed(MStoFD) mode.

PUBB in the MS mode has a single subproblem pool and the tree search is controlled by the centralized scheme. In this mode, subproblems are transferred between **PM** and **Solver** every time after **Solver** finishes an "evaluation". In PUBB, the evaluation of a subproblem consists of the following steps: (1)obtain a subproblem from the system, (2)compute a bound value (**UPPER_BOUNDING**), (3)generate new subproblems (**BRANCHING**) if necessary, and (4)remove the current subproblem. In our computing environment, the mean transfer time of a subproblem took more than 0.01 seconds, but the mean evaluation time of a subproblem for the maximum clique problem took less than 0.002 seconds. We cannot expect to obtain good performance for this kind of problems with a fine grained evaluation procedure in this mode.

PUBB in the FD mode has multiple subproblem pools and the tree search is controlled by a fully distributed scheme. In this mode, load balancing of subproblems is needed, and PUBB adopts a pairwise balancing strategy (see [9]). In **Solver**, a sequential branch-and-bound algorithm is performed. Each **Solver** notifies the state of its subproblem pool to the corresponding **LB** every time after an evaluation procedure is done. Hence, each **LB** recognizes the state of the corresponding **Solver** and, based on the state, the **LB** searches the partner to send/receive subproblems among other **LB**s. Usually, **LB** begins to search when the number of subproblems in the corresponding **Solver** is less than the *threshold value*, which is a parameter, and the search is performed independently from the underlying evaluation procedure in **Solver**s. Subproblems are transferred between **Solver**s only when a subproblem pool of some **Solver** is almost empty (the number of subproblems is less than a threshold value) or filled up to its capacity. A sender side **Solver** having many subproblems is selected, and the selected sender **Solver** transfers one subproblem for every one that it evaluates. Since the transfer between two **Solver**s with non-blocking send is performed every once after the evaluation procedure in the sender side is done, the transfer process overlaps with the evaluation procedure. Therefore, there are possible modifications to obtain better performance. The modification is described in the next subsection.

PUBB in the MStoFD mode changes the internal archi-

tecture and control scheme from the MS mode to the FD mode. In this mode, PUBB begins with the MS mode and runs until the number of subproblems maintained by the **PM** becomes greater than the value specified as a parameter. After that, PUBB starts changing the internal architecture to the same as that in the FD mode. When the **PM** recognizes that the change is finished, it selects a subproblem from the pool with best bound order and distributes it to different **Solver** one by one using "round robin" schemes.

3.2 Adjustment to Applications with a Fine Grained Evaluation Procedure

In this subsection, we attempt to improve performance in the FD mode to adjust PUBB to applications with a fine grained evaluation procedure. When the evaluation is fine grained, the communication costs between **Solver** and the corresponding **LB** cannot be neglected even if it is the interprocess communication within a workstation. In order to control the communication costs, a parameter which specifies notification interval time was added to PUBB. **Solver** does not send states of its subproblem pool to the corresponding **LB** until the time elapsed from the previous notification exceeds the specified interval time, except when the following situations occur: (1)Subproblem pool in the **Solver** is almost empty. (2)**Solver** is sending/receiving subproblems. In these two situations, **LB** should start/stop pairwise load balancing as soon as possible and it is kicked when the **LB** recognizes the necessity to start/stop from the notification. Running PUBB with the notification interval time is illustrated in Figure 3.

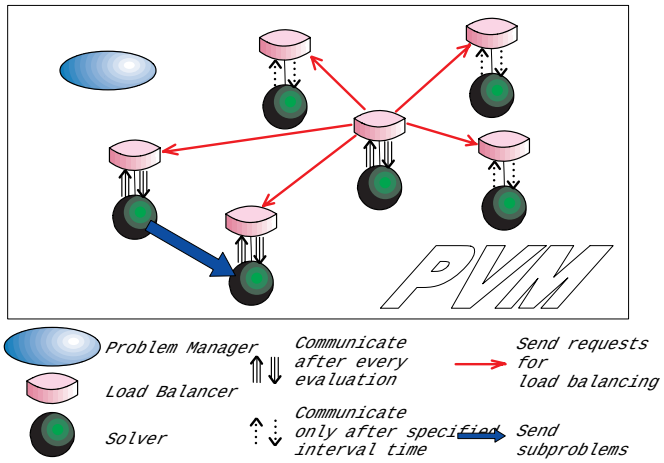

Figure 3. Running PUBB

When the interval time is specified, each **LB** must recognize the states of the corresponding **Solver** from delayed notifications. If a long interval time is specified, the communication cost between **LB** and **Solver** decreases. However, if the interval time is too long, the states maintained in **LB** does not reflect the states of the corresponding **Solver** at all and can mislead load balancing procedure. This situation might make larger overhead costs than the declined communication costs.

In order to set the interval time appropriately, computational experiments with 20 Solvers were conducted. All computational experiments with parallel run in this paper were conducted under the following environment and parameter setting. Only one **Solver** was run on each workstation and the **PM** was hosted by a workstation where no **Solver** and no other user processes ran. According to our previous results[9], the MStoFD mode was applied and the control scheme was changed when the number of subproblems maintained by the **PM** exceeded the following number: $(\text{threshold value}+1) \times (\text{the number of } \textbf{Solver}\text{s})$. In all experiments with parallel run in this section, five runs were repeated with the same parameter specifications in the same computing environment to assess the validity of this parameter setting and modifications of PUBB.

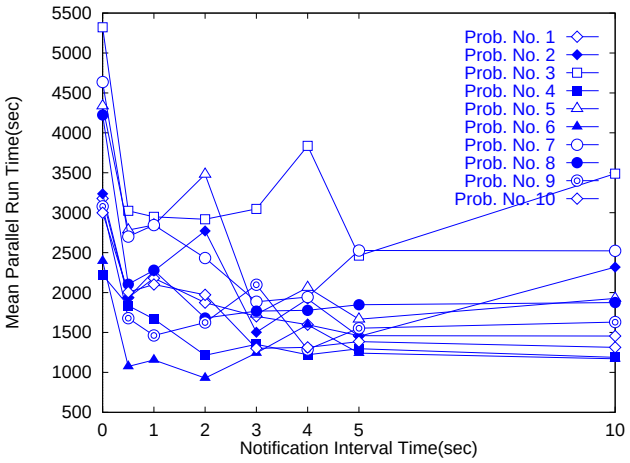

Figure 4. Mean parallel run time in the different notification interval with 20 Solvers

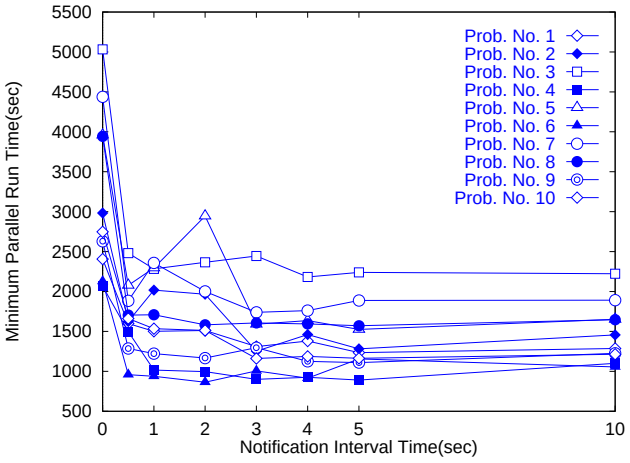

Figure 5. Minimum parallel run time in the different notification interval with 20 Solvers

Figures 4 and 5 show the mean and minimum parallel run time, respectively. These figures show that the interval time is essential to decrease the parallel run time. The parallel run time always decreases when the interval time is more than zero and decreases more than half of that when the interval time is equal to zero (that is, notification is done every after evaluation) in several cases. Figure 4 shows that the mean parallel run time appears to decrease until one second interval time and after that it greatly varies in many cases. Figure 5 shows that the minimum parallel run time does not decrease in many cases even if the interval time takes longer than one second. Therefore the interval time was set to one second.

3.3 Depth First Tree Search with Best Bound First Transfer

PUBB supports several tree search strategies. When the depth first search is specified in the FD mode, not only selection of a subproblem for the next evaluation, but also the transferring to the other **Solver** is carried out in the depth first order. Usually, a subproblem sent from the sender side **Solver** is evaluated earlier in the FD mode, because the receiver side **Solver** often has very small amount of subproblems.

However, this way of transferring should have heavy overhead. The transferred subproblem may not yield many new subproblems. This can be improved by transferring to the other **Solver** in the best bound first order, while the depth first tree search is still applied in the sequential branch-and-bound algorithm in **Solver**. Table 2 shows the effect of this improvement. In the table, D.F.T. abbreviates Depth First Transfer and B.F.T. abbreviates Best bound First Transfer. The improvement is quite dramatic in reduction of the number of transfered subproblems. Hence, it is also dramatic in cutting down the parallel run time. This modification shortens the parallel run time even more than that we obtained in the previous subsection.

Table 2. Computational results in different transfer strategy

Prob. No.	Strategy	Parallel Run Time(sec)			# of Transferred Subproblems		
		Mean	Min	Max	Mean	Min	Max
1	D.F.T.	2179.28	1501.61	2884.47	259991	170106	327651
	B.F.T.	1006.08	906.00	1372.75	13631	10511	15794
2	D.F.T.	2269.64	2017.45	2562.68	252986	198468	321987
	B.F.T.	1082.22	1077.04	1092.12	14443	12409	16685
3	D.F.T.	2947.66	2283.82	4071.38	328107	170455	488439
	B.F.T.	1950.58	1933.30	1974.37	20101	17504	22769
4	D.F.T.	1668.40	1015.86	1968.98	172703	90678	237112
	B.F.T.	821.63	809.89	831.06	14049	10683	16617
5	D.F.T.	2845.97	2295.11	3652.39	380516	233904	480927
	B.F.T.	1468.25	1452.99	1492.37	18750	15331	23181
6	D.F.T.	1156.16	938.65	1575.85	120232	79115	200219
	B.F.T.	769.58	751.97	796.58	11766	9511	14708
7	D.F.T.	2845.39	2357.35	3436.10	331042	262169	419683
	B.F.T.	1570.68	1548.15	1599.40	17451	13979	25608
8	D.F.T.	2279.22	1709.44	2502.05	231953	96859	369319
	B.F.T.	1462.07	1445.58	1492.34	15378	12237	17268
9	D.F.T.	1462.18	1222.38	1611.45	96955	59047	145142
	B.F.T.	1214.67	956.86	1359.50	10862	7358	16427
10	D.F.T.	2100.77	1534.42	2725.48	223286	147135	287155
	B.F.T.	1073.00	1055.16	1099.78	11772	8500	17259

3.4 Speedup and its Estimation

The *speedup* is defined as follows:

$$\text{Speedup}(x) = \frac{\text{Computing time in the sequential algorithm}}{\text{Parallel run time with } x \textbf{ Solver}\text{s}.}$$

For the discussion of speedup related to parallel branch-and-bound algorithms, see, *e.g.* [6].

Figure 6 shows the speedup. In this figure, all the results are plotted using the mean value of five repeated runs. In the present case, as the STABJOIN is a very strong heuristic, an optimal solution is often found in the initial solution of the algorithm. Therefore, for many problem instances, computing power was used for the certification of its optimality. In such a case, the total amount of work is exactly the same as the sequential one and the anomalies cannot occur, and the speedup is always under linear with parallel overhead. Parallel run time is measured as real time (the total wall clock our program took to load, execute, and exit), that is, it includes several kinds of overheads caused by the system, but computing time in the sequential algorithm is measured as user time (the total amount of CPU time our program took to execute), that is, it does not include any of these. Therefore, this is a very strict measurement of speedup. Even so, the results show good performance. Instances which improved solution(s) during the tree search achieved better speedup than those which did not improve. Especially, in some cases, super linear speedup was achieved.

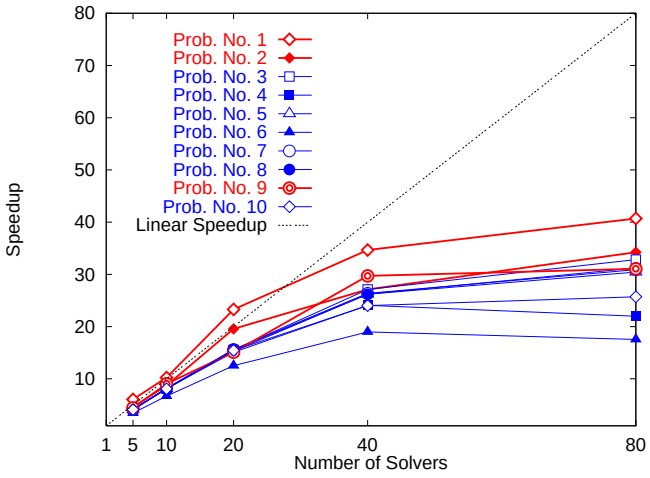

Figure 6. Speedup as a function of the number of Solvers

In order to estimate how much speedup is obtained in only one trial of parallel run, we introduced the following estimation of speedup:

$$\text{Est\_Speedup}(x) = \frac{\text{Total CPU time consumed for evaluation procedure with } x \textbf{ Solver}\text{s}}{\text{Parallel run time with } x \textbf{ Solver}\text{s}}$$

Est_Speedup gives a lower bound of speedup theoretically, in the case that the calculation starts with an optimal solu-

tion. Table 3 shows this estimation and the real speedup for every trial in the two problem cases. One (Prob. No. 3) is a case which starts with optimal solution, the other (Prob. No. 1) is the opposite. When the initial solution is optimal, speedup is estimated fairly accurately. On the other hand, when the initial solution is not optimal, the value does not make sense. However, in our results, the estimation is usually much lower than the actual speedup.

Table 3. Estimation of speedup

Prob. No.	# of Solvers	Estimation[Actual]				
1	5	3.8 [6.0]	3.8 [6.0]	3.8 [6.0]	3.8 [6.1]	3.8 [6.1]
	10	7.5 [9.3]	7.5 [13.5]	7.5 [10.2]	7.5 [8.5]	7.5 [11.0]
	20	14.0 [25.9]	14.1 [25.5]	13.8 [25.5]	14.5 [17.1]	13.9 [25.7]
	40	25.1 [29.8]	20.4 [37.3]	22.4 [41.0]	25.3 [28.8]	22.0 [40.4]
	80	20.4 [36.6]	19.6 [35.3]	24.4 [44.5]	24.7 [44.6]	25.0 [44.8]
3	5	3.8 [4.1]	3.8 [4.1]	3.8 [4.1]	3.8 [4.1]	3.8 [4.1]
	10	7.5 [8.1]	7.5 [8.1]	7.5 [8.1]	7.5 [8.1]	7.5 [8.1]
	20	14.4 [15.6]	14.2 [15.3]	14.3 [15.5]	14.5 [15.7]	14.5 [15.6]
	40	25.6 [27.4]	25.1 [26.8]	24.9 [26.7]	26.0 [27.9]	25.1 [26.9]
	80	33.2 [35.2]	31.5 [33.4]	28.1 [29.6]	28.3 [30.2]	34.9 [36.9]

4 Computational Results for Challenging Problem Instances

Finally, we report exact solutions of five problem instances from the second DIMACS Challenge. To the best of our knowledge, they have not been solved exactly yet. Table 4 displays data and the exact solution (maximum clique) of the problem instances.

Table 5 shows the result of the parallel branch-and-bound algorithm with 50 solvers. By the result of C250.9, we know that structured problems are often hard to solve exactly and the fact calls for other upper bounding procedures. As shown in the column "# of Improved Solutions", STABJOIN produced the exact solution in four of the five problem instances, which means that STABJOIN is promising also for structured problem instances. Quite a good speedup could be obtained from the estimation.

Table 4. Exact Solutions for the DIMACS Problem Instances

C250.9 (# of vertices = 250, density = 89.9%, maximum clique size = 44)
3, 6, 8, 10, 20, 26, 27, 29, 30, 35, 37, 48, 51, 55, 63, 64, 66, 84, 87, 90, 92, 97, 102, 105, 108, 111, 129, 138, 142, 147, 152, 153, 157, 160, 163, 177, 185, 197, 198, 212, 232, 235, 241, 243
p_hat500-3 (# of vertices = 500, density = 75.2%, maximum clique size = 50)
19, 23, 39, 63, 69, 102, 123, 137, 150, 171, 173, 184, 186, 194, 206, 215, 221, 222, 231, 248, 259, 266, 269, 282, 300, 302, 320, 323, 342, 348, 349, 368, 370, 381, 383, 404, 408, 412, 418, 428, 440, 445, 455, 460, 465, 475, 480, 489, 490, 500
p_hat700-3 (# of vertices = 700, density = 74.8%, maximum clique size = 62)
8, 16, 17, 30, 31, 54, 76, 87, 106, 115, 122, 127, 129, 142, 145, 149, 152, 193, 239, 240, 248, 261, 276, 285, 290, 292, 295, 306, 317, 323, 325, 326, 345, 348, 350, 361, 368, 373, 379, 387, 393, 459, 468, 482, 492, 495, 511, 525, 537, 562, 579, 581, 584, 587, 608, 612, 623, 638, 649, 653, 661, 678
p_hat1000-2 (# of vertices = 1000, density = 49.0%, maximum clique size = 46)
8, 65, 84, 92, 155, 227, 253, 271, 283, 284, 298, 322, 330, 370, 378, 464, 489, 492, 495, 497, 537, 538, 549, 562, 600, 608, 623, 630, 636, 639, 640, 653, 671, 777, 782, 798, 800, 801, 819, 824, 862, 864, 900, 961, 993, 997
p_hat1500-2 (# of vertices = 1500, density = 50.6%, maximum clique size = 65)
1, 68, 72, 73, 131, 133, 166, 173, 214, 261, 290, 358, 389, 392, 403, 423, 428, 434, 438, 440, 444, 450, 514, 529, 555, 561, 562, 564, 606, 622, 689, 691, 723, 738, 749, 767, 819, 838, 863, 878, 906, 917, 920, 940, 960, 974, 990, 1005, 1034, 1039, 1069, 1072, 1145, 1167, 1191, 1209, 1247, 1265, 1267, 1333, 1385, 1411, 1415, 1478, 1483

Table 5. Computational results

Data Name	Comp. Time (sec)	Speedup Estimation	# of Generated Subproblems	# of Evaluated Subproblems	# of Improved Solutions
C250.9	25768.79	38.1	687551803	687551803	0
p_hat500-3	856.65	34.5	12398047	12398047	0
p_hat700-3	10637.92	43.1	131620573	131620573	0
p_hat1000-2	1111.23	35.7	16098000	16098000	0
p_hat1500-2	61694.40	45.2	628386474	628386468	1

5 Concluding Remarks

We undertook solving the maximum clique problem, using PUBB and exploiting the latest results. We first implemented the sequential algorithm, which behaved best with the very fast upper bounding (evaluation) procedure. PUBB possesses an easy translation from an existing sequential algorithm to a parallel one. On the other hand, in our computing environment, PUBB has been suitable for applications with coarse grained evaluation procedure, because it runs on networked workstations connected with the slow Ethernet. Therefore, we modified PUBB to adjust the application with the fine grained evaluation procedure and showed the effectiveness for such applications. As a result, PUBB becomes very promising in wide variety of problems, regardless of the granularity of the evaluation procedure.

References

[1] E. Balas and J. Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15:397–412, 1996.

[2] M. Benaïchouche, V. D. Cung, S. Dowaji, B. L. Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In A.Ferreira and P.Paradalos, editors, *Solving Combinatorial Optimization Problems in Parallel*, volume 1054 of *Lecture Notes in Computer Science*, pages 201–231. Springer-Verlag, 1996.

[3] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[4] A. Brunggger, A. Marzetta, J. Clausen, and M. Perregaard. Joining forces in solving large-scale quadratic assignment problems in parallel. In *Proc. of the 11th International Parallel Processing Symposium*, pages 418–427. IEEE Computer Society Press, 1997.

[5] T. Ikebe and A. Tamura. Ideal polytopes and face structures of some combinatorial optimization problems. *Mathematical Programming*, 71:1–15, 1995.

[6] T. H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27:594–602, 1984.

[7] C. Manino and A. Sassano. An exact algorithm for the maximum stable set problems. *Computational Optimization and Applications*, 3:243–258, 1994.

[8] P. Pardalos and J. Xue. The maximum clique problem. *Journal of Global Optimization*, 4:301–328, 1994.

[9] Y. Shinano, K. Harada, and R. Hirabayashi. Control schemes in a generalized utility for parallel branch-and-bound algorithms. In *Proc. of the 11th International Parallel Processing Symposium*, pages 621–627. IEEE Computer Society Press, 1997.

[10] S. Tschöke and T. Polzer. Portable parallel branch-and-bound library(ppbb-lib): User manual version 1.1. Technical report, University of Paderborn, 1996.