EFFICIENT METHODS FOR ASYNCHRONOUS DISTRIBUTED CONSTRAINT OPTIMIZATION ALGORITHM

Toshihiro Matsui, Hiroshi Matsuo and Akira Iwata
Department of Electrical and Computer Engineering
Nagoya Institute of Technology
Gokiso-cho,Showa-ku,NAGOYA,466-8555,JAPAN
email: {tmatsui@mars.elcom.,matsuo@,iwata@}nitech.ac.jp

ABSTRACT
Effective methods to reduce number of message cycles of adopt algorithm which is a complete algorithm for distributed constraint optimization problems are proposed. The Adopt algorithm can perform branch and bound search asynchronously. However, it has overhead in backtracking and searches same partial solution repeatedly. To reduce overhead of backtracking, lower and upper bound of cost of partial solution are considered and some messages are sent to upper nodes by shortcut. Leaning of the lower and upper bound is used to reduce extra search. The results show the efficiency of the proposed methods.

KEY WORDS
Constraint optimization/satisfaction problem, Distributed search, Branch and bound, Multi agent system, Distributed Artificial Intelligence,

1 Introduction

Distributed constraint optimization problem (DCOP) [5][6] is a constraint optimization problem (COP) [1][2] which is applied to distributed search. DCOP is an important area of researches of multi-agent system because it formalizes optimization problem which cannot be formalized by distributed constraint satisfaction problem (DCSP) [3].

Branch and bound search methods for over-constrained CSPs are presented in [2]. However, most existing methods for DCOP are based on DCSP algorithm or approximated algorithm [4][5]. Adopt [7] is a complete algorithm for DCOP based on branch and bound search. In this algorithm, nodes (agents) are priorized in depth first search tree for cost computation. The nodes perform search asynchronously.

However, adopt algorithm has overhead in backtracking and searches same partial solution repeatedly due to asynchronisity and memory restriction. In this paper, methods to reduce these overheads are presented. To reduce overhead of backtracking, lower and upper bound of cost of partial solution are considered and some messages are sent to upper nodes by shortcut. Leaning of the lower and upper bound is used to reduce extra search.

2 DCOP and Adopt algorithm

First, we illustrate DCOP and Adopt algorithm [7] in brief. In this paper, some notation and rules are modified. The DCOP as following is considered. A node (agent) i has a variable x_i. x_i has a value of $d_i \in D_i$. Variables are related by binary constraint. Let x_j denote a variable of node j which is related to x_i. Cost of the value pair $((x_i, d_i), (x_j, d_j))$ is evaluated by $f_{ij}(d_i, d_j)$ which is the cost function of the constraint. Each node knows the constraints and cost functions which are related to their own variable. Each node selects the value of its own variable. In the following, node i and variable x_i can be used interchangeably. Cost of a solution is a conjunction of $f_{ij}(d_i, d_j)$ for all constraints. Optimal solution has the minimum cost. Each node has message communication links to another nodes. Order of messages in a link is not changed. However, order of messages between links is not kept.

In adopt algorithm, it is assumed that nodes are priorized in a depth first search (DFS) tree of a constraint network. There are no constraints between subtrees of DFS tree. Therefore each subtree has parallelism. A node i knows $parent_i$, set of children $children_i$ and set of upper / lower neighborhoods which are related by constraints $upperNeighbors_i$ / $lowerNeighbors_i$.

Execution of the algorithm is as following. (1) Each node evaluates the cost of the current solution and the cost allocation. The node selects the value of its variable according to the evaluation. The value is notified to the lower neighborhoods which are related by constraints (VALUE message). (2) Each node notifies the cost of the current solution to its parent (COST message). (3) Each node decides the cost allocation between itself and its children. The cost allocation is notified to children (THRESHOLD message). After repeating the above process, the lower and upper bound of evaluated cost become equal in the root node. The root node then selects the optimal value of its variable and notifies termination to children (TERMINATE message). The children search the optimal value of its variables and terminate. Finally, all nodes terminate and its allocated values are the optimal solution. The components of the adopt algorithm are shown in the following subsection.

2.1 Variables

Node i has following variables.

- d_i: value of variable x_i
- $threshold_i$: backtracking threshold
- $currentContext_i$: solution of upper nodes
- $context_i(d,x)$: solution of upper nodes of a child
- $lb_i(d,x)$,$ub_i(d,x)$: upper and lower bound of cost of solution of a child
- $t_i(d,x)$: backtracking threshold for a child

d_i is changed according to cost evaluation and is notified to lower neighborhoods by VALUE messages. $threshold_i$ is the cost which is allocated to subtree by its parent. It is used to determine whether to change the value of d_i. $currentContext_i$ is cache of solution of upper nodes. When VALUE or COST messages are received, $currentContext_i$ is updated. $context_i(d,x)$, $lb_i(d,x)$ and $ub_i(d,x)$ are cache of cost for $d \in D_i$, $x \in children_i$. They are updated when COST message is received. $t_i(d,x)$ is set by node i and it is notified to children by THRESHOLD messages. Initial values of the variables are following: $d_i \in D_i$, $threshold_i = 0$, $currentContext_i = \{\}$, $context_i(d,x) = \{\}$, $lb_i(d,x) = 0$, $ub_i(d,x) = \infty$, $t_i(d,x) = 0$.

2.2 Definition of Costs

For $d \in D_i$, local cost $\delta_i(d)$ of node i is defined as the sum of cost of constraints for upper neighborhoods. For $d \in D_i$, lower bound $LB_i(d)$ and upper bound $UB_i(d)$ of cost of the subtree rooted at i are defined as the sum of local cost and lower or upper bound for all children. Lower bound LB_i and upper bound UB_i of the subtree are the minimum value of $LB_i(d)$ and $UB_i(d)$ for $d \in D_i$.

$$\begin{aligned}
\delta_i(d) &= \sum_{\substack{(x_j,d_j)\in currentContext_i,\\ x_j \in upperNeighbors_i}} f_{i,j}(d,d_j)\\
LB_i(d) &= \delta_i(d) + \sum_{x\in children_i} lb(d,x)\\
UB_i(d) &= \delta_i(d) + \sum_{x\in children_i} ub(d,x)\\
LB_i &= min_{d\in D_i} LB_i(d)\\
UB_i &= min_{d\in D_i} UB_i(d)
\end{aligned}$$

2.3 Conditions for variables

Node i maintains the following conditions.
[condition for $context_i(d,x)$]

$$\forall d \in D, x \in children,\\ context_i(d,x) \ and \ currentContext_i \ are \ compatible$$

If $context_i(d,x)$ and $currentContext_i$ are incompatible, $lb_i(d,x)$, $ub_i(d,x)$ and $t_i(d,x)$ which are related to $context_i(d,x)$ are also incompatible with $currentContext_i$. Therefore $context_i(d,x)$ and $currentContext_i$ must be compatible. It means that the values of same variables in both contexts must be equal. If they are incompatible, $context_i(d,x)$, $lb_i(d,x)$, $ub_i(d,x)$ and $t_i(d,x)$ are re-initialized to their initial values.
[condition for $t_i(d,x)$ (ChildThresholdInvariant)]

$$\forall d \in D, x \in children_i,\\ lb_i(d,x) \le t_i(d,x) \le ub_i(d,x)$$

Backtracking threshold for a child must not exceed the lower and upper bound of cost of the child. $t_i(d,x)$ is limited by the lower or upper bound.
[condition for $threshold_i$ (ThresholdInvariant)]

$$LB_i \le threshold_i \le UB_i$$

Backtracking threshold for node i must not exceed the lower and upper bound of cost of the node. $threshold_i$ is limited by the lower or upper bound.
[condition for d_i]

$$\begin{cases} UB_i(d_i) = UB_i & if \ \ threshold_i = UB_i\\ LB_i(d_i) \le threshold_i & otherwise \end{cases}$$

Lower bound of cost for value of variable must not exceed backtracking threshold. If it exceeds threshold, value d_i is changed as $LB_i(d_i) = LB_i$. This backtracking strategy is considered as an optimistic search. Especially, if backtracking threshold and upper bound are equal, the value for the upper bound must be selected. This condition is needed to select optimal solution.
[condition for $t_i(d,x)$ (AllocationInvariant)]

$$threshold_i = \delta_i(d_i) + \sum_{x\in children_i} t_i(d_i,x)$$

Backtracking threshold is a cost which is allocated to subtree rooted at the node. Therefore it must be equal to sum of local cost and backtracking threshold for all children. If they are different, $t_i(d,x)$ are adjusted. [condition for $t_i(d,x)$ (ChildThresholdInvariant)] must also be kept.

2.4 Message sending

Node i sends the following messages to the other nodes [1] .
[send (VALUE,(x_i,d_i))]

condition to send: d_i has updated
destination: $x \in lowerNeighbors_i$

[send (COST,x_i,$currentContext_i$,LB_i,UB_i)]

[1] In [7], Conditions for message sending is not described strictly. We use a set of conditions which can perform the algorithm correctly.

```
    condition to send:
    (VALUE messages are received from $\forall x \in upperNeighbors_i$) $\wedge$ ($d_i$, $currentContext_i$,
    $threshold_i$ or the costs of children are updated)
    $\wedge$ (TERMINATE message is not received)
    destination: $parent_i$
[send (THRESHOLD,$currentContext_i$,$t_i(d_i, x)$)]
    condition to send:
    $d_i$, $currentContext_i$ or the costs of children are
    updated
    destination: $x \in children_i$
[send (TERMINATE,$currentContext_i \cup (x_i, d_i)$)]
    condition to send:
    condition for termination is true (shown later)
    destination: $x \in children_i$
```

2.5 Message receiving

Node i processes the followings, when messages are received.

```
[when (VALUE,$(x_k, d_k)$) is received]
    if TERMINATE is not received then
      add $(x_k, d_k)$ to $currentContext_i$ (or replace).
    endif
[when (COST,$x_k$,$context_k$,$lb_k$,$ub_k$) is received]
    $d \leftarrow d$ that $(x_i, d) \in context_k$
    remove $(x_i, d)$ from $context_k$.
    if TERMINATE is not received then
      $\forall (x_j, d_j) \in context_k$,
      $x_j \notin upperNeighbors_i$,
        add $(x_j, d_j)$ to $currentContext_i$
        (or replace).
    endif
    maintain [condition for $context_i(d, x)$].
    if $context_k$ and $currentContext_i$
      are compatible then [2]                    (2.5)
      $\forall (x_j, d_j) \in context_k$,
        add $(x_j, d_j)$ to $context_i(d, x_k)$ (or replace).
      if $lb_k > lb_i(d, x_k)$ then
        $lb_i(d, x_k) \leftarrow lb_k$ endif
      if $ub_k < ub_i(d, x_k)$ then
        $ub_i(d, x_k) \leftarrow ub_k$ endif
    endif
[when (THRESHOLD,$context_k$,$t_k$) is received]
    if $context_k$ and $currentContext_i$
      are compatible then $threshold_i \leftarrow t_k$ endif
[when (TERMINATE,$context_k$) is received]
    $currentContext_i \leftarrow context_k$
    record that TERMINATE is received.
```

2.6 Optimal cost and termination of search

In root node i, $LB_i = threshold_i = UB_i$ will eventually occur. Then i fixes its value to optimal solution and terminates. When parent node of node j is terminated and $threshold_j = UB_j$ occures, j fixes its value to optimal solution and terminates.

[Termination detection]

```
if $threshold_i = UB_i \wedge$
  ($i$ is root node $\vee$ TERMINATE is received)
  then
   ($UB_i(d_i) = UB_i$ from [condition for $d_i$])
   record that condition for termination is true.
endif
```

[2]In this paper, following part of this procedure is modified. By this modification, even if a COST message which has compatible $context_k$, $lb_k < lb_i(d, x_k)$ and $ub_i(d, x_k) < ub_k$ is received, the monotonisity of cost is guaranteed.

2.7 Total flow of processing

The order of processings for above conditions and messages is not unique. In our implementation, each node repeats the processes of receiving messages, maintaining conditions and sending messages.

3 partial solution for upper/lower bound and shortcut of message sending

In above algorithm, nodes perfom backtracking asynchronously. However this simple backtracking has the same redundancy as that of ordinary backtracking algorithm. In this section, we present a method to reduce the overhead of backtracking. Partial solutions for upper/lower bound are derived and COST messages are sent by shortcut. We assume that (1) each node has knowledge of path form the node to root node, (2) each node has arbitrary links which do not follow constraint edges, and (3) messages sent to terminated nodes are ignored.

3.1 partial solution for upper/lower bound

In adopt algorithm, node i has $context_i(d, x)$ which is a (parital) solution of upper nodes of a child. This solution proves the values of $lb_i(d, x)$ and $ub_i(d, x)$. However, parital solution needed to prove lower bound may not contain all allocations of values of upper nodes. Therefore $context_i(d, x)$ is separated into $context_i^{lb}(d, x)$ and $context_i^{ub}(d, x)$ which are least solutions derived to prove $lb_i(d, x)$ and $ub_i(d, x)$. This concept is similar to resolvant-based learning of DCSP [8]. If the partial solution for lower bound does not contain value of parent, cost notification is sent by shortcut to reduce redundancy of backtracking. This is similar to backjumping in COP [2].

3.2 Derivation of partial solution

For $X'_d \subseteq upperNeighbors_i$ and $X''_d \subseteq children_i$,if

$$\forall d \in D_i,$$
$$LB_i \leq \sum_{\substack{x_j \in X'_d, (x_j, d_j) \in \\ currentContext_i}} f_{i,j}(d, d_j) + \sum_{x \in X''_d} lb_i(d, x)$$

then partial solution $context_i^{lb}(d, x)$ which proves lower bound is as following.

$$\bigcup_{d \in D_i} \quad (\{(x_j, d_j) \in currentContext_i | x_j \in X'_d\} \cup \bigcup_{x \in X''_d} context_i^{lb}(x, d))$$

To select a (X'_d,X''_d) pair, certain evaluations such as, (1) size of sets is small or (2) only upper variables are contained, should be considered. In this paper, we choose the latter. For each $d \in D_i$, a (X'_d,X''_d) pair whose lowest variable is higher than that in other pairs is selected.

Partial solution $context^{ub}_i(d, x)$ which proves upper bound depends on all constraints related to the subtree rooted at node i.

$$\{(x_j, d_j) \in currentContext_i | \\ x_j \in upperNeighbors_i\} \cup \\ \bigcup_{d \in D_i, x \in children_i} context^{ub}_i(x, d)$$

Therefore $context^{ub}_i(d, x)$ is almost equal to *current-Context*$_i$ if, delay of message and resetting of cache of the child x, are not considered.

3.3 Modification of COST message

Format and destination of the COST message is modified to send information of the separate partial solutions for upper and lower bound. COST message sent to the parent includes partial solution for upper and lower bound.

(COST,x_i,$currentContext_i$,
$context^{lb}_i$,$context^{ub}_i$,LB_i,UB_i)

If $context^{lb}_i$ does not contain the variable of the parent node, additional COST message for lower bound is sent by shortcut. However, if $LB_i = 0$, it is clearly unnecessary to send. The destination node j has lowest variable x_j of $context^{lb}_i$. The variable x_k of the message is variable of a node, which is a child of j and an ancestor of i. UB_i is not proven by shortcut message, therefore default upper bound ∞ and solution $\{\}$ are used. The COST message is as follows.

(COST,x_k,$context^{lb}_i$,
$context^{lb}_i$,$\{\}$,LB_i,∞)

In message receiving [when received (COST, x_k, $context_k$, lb_k, ub_k)], the part from (2.5) is modified to record partial solution for upper and lower bound separately. The modified part for lower bound is as follows.

```
**if** $context_k$ and $currentContext_i$ is compatible
  **then**
  **if** $context^{lb}_k$ contains $(x_i, d')$ **then**
    $d' \leftarrow d'$ that $(x_i, d') \in context^{lb}_k$
    remove $(x_i, d')$ from $context^{lb}_k$.
  **endif**
  for $d = d'$ if $context^{lb}_k$ contained $(x_i, d')$,
   otherwise $\forall d \in D_i$
    **if** $lb_k > lb_i(d, x_k)$ **or** ($lb_k = lb_i(d, x_k)$
    **and** $|context^{lb}_k| < |context^{lb}_i(d, x_k)|$)
    **then**
     $lb_i(d, x_k) \leftarrow lb_k$
     $context^{lb}_i(d, x_k) \leftarrow context^{lb}_k$
    **endif**
**endif**
```

Similarly, the part for upper bound is modified such that $ub_i(x, d)$ decreases monotonously.

[condition for $context_i(d, x)$] is also separated for upper and lower bound. $t_i(d, x)$ is pushed up by $lb_i(d, x)$. Therefore $t_i(d, x)$ is reset with $lb_i(d, x)$ when $context^{lb}_i(d, x)$ is incompatible.

4 Learning of upper/lower bound

One of the merits of the adopt algorithm is its polynomial memory. Each node i records a set of $context_i(d, x)$, $lb_i(d, x)$ and $ub_i(d, x)$. If they are incompatible with $currentContext_i$, they are reset to their initial value. However, the reset causes an increase in search cycle. If enough memory is available, the partial solutions should be recorded in order to reduce the number of cycles. In this section, learning of partial solutions is presented. This is similar to Nogood learning of CSP/DCSP [9][10].

4.1 Adding of cache

If all $context^{lb}_i(d, x)$ or $context^{ub}_i(d, x)$ and corresponding $lb_i(d, x)$ or $ub_i(d, x)$ (denoted by $context^{lb/ub}_i(d, x)$ and $lb/ub_i(d, x)$ in the following) are recorded, no reset of cache occurs. However, that is impossible if the problem size is not small, because it requires exponential memory. Therefore, we use a set of buffers which are managed by LRU. Buffers which have a finite length are added for each $context^{lb/ub}_i(d, x)$ and first element of the buffer is used as $context^{lb/ub}_i(d, x)$. If the first element and $currentContext_i$ is incompatible, the buffer is maintained. If compatible element is in the buffer, the element is moved to front. If it is not found, a new element is inserted at front. If length of the buffer is over the limitation, the last element is removed. The corresponding $lb/ub_i(d, x)$ are also managed.

4.2 Integration of partial solutions

If only one $context^{lb/ub}_i(d, x)$ (and $lb/ub_i(d, x)$) compatible with $currentContext_i$ is recorded in the buffer, the context is modified by cost notification and previous compatible context is forgotten. It is reasonable to record compatible contexts which partially overlap in solution space. Therefore, those partial solutions are recorded in the buffer. If a partial solution is included by another, solutions are integrated to reduce memory. If

$$lb \geq lb' \wedge context^{lb} \subseteq context'^{lb} \\ ub \leq ub' \wedge context^{ub} \subseteq context'^{ub}$$

then $context'^{lb/ub}$ and lb'/ub' are replaced by $context^{lb/ub}$ and lb/ub. In other case, both of them are recorded.

When COST messages are received, partial solutions are added or replaced according to above conditions. In cost evaluation, compatible $context'^{lb/ub}$ and corresponding lb'/ub' which have most narrow boundary are selected as first element of caches.

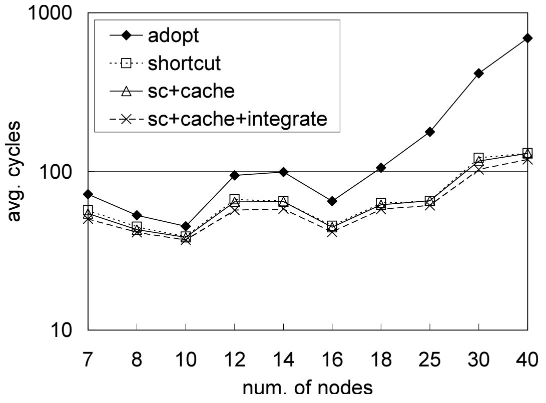

Figure 1. Average number of cycles to find the optimal solution (weight of constraint=1,d=2)

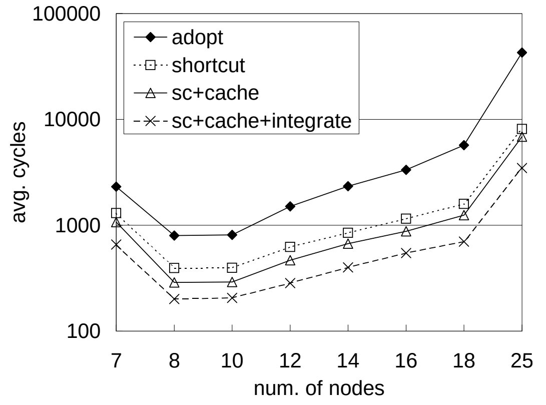

Figure 2. Average number of cycles to find the optimal solution (weight of constraint=1,d=3)

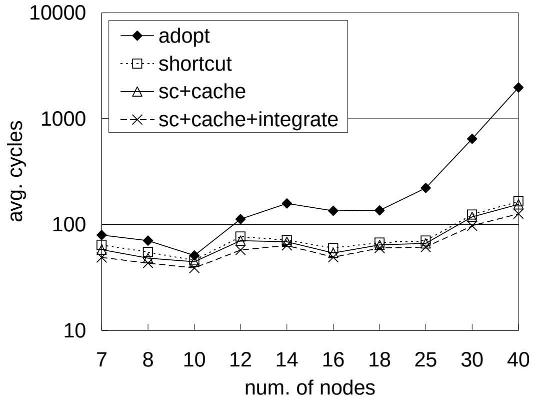

Figure 3. Average number of cycles to find the optimal solution (weight of constraint=$\{1, \cdots, 10\}$,d=2)

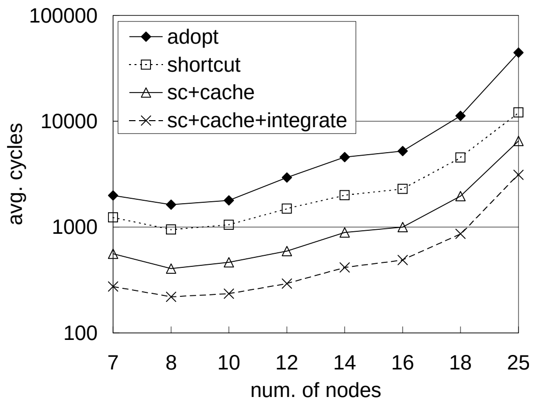

Figure 4. Average number of cycles to find the optimal solution (weight of constraint=$\{1, \cdots, 10\}$,d=3)

5 Evaluation

The results of simulations are shown above. We use graph coloring problem with 3 colors which is experimented in [7]. For n number of nodes and parameter $d = \{2, 3\}$, the number of constraint is determined as $n \cdot d$. Weight of constraints are 1 or $\{1, \cdots, 10\}$. Normal adopt algorithm (adopt), adopt using shortcut message (shortcut), shortcut using additional context caches to learn partial solutions (sc+cache), and sc+cache using context integration (sc+cache+integrate) are evaluated. Length of caches is 100 for each (d, x).

First, we simulate distributed system. The system performs cycles of message exchange and processing of nodes. The node performs message receiving, maintaining of conditions and message sending in each cycle. Results of 25 problems are averaged. Average number of cycles to find optimal solution are shown in Fig. 1, 2, 3, 4. In the case of $d = 2$, the effect of learning partial solutions is less than the effect of shortcut message. It can be considered that the algorithm does not search same solution repeatedly in less constrained problem. In the case of $d = 3$, the number of cycles are reduced by learning partial solutions. Average number of messages per cycle is shown in Fig. 5. In methods using shortcut messages, the number of messages increases. However, the increment of total messages is not large.

To evaluate the performance of the algorithm in asynchronous distributed environment, we use the MPI environment. For this experiment, the program which is used in above simulation is modified to send and receive message using MPI. The performance tuning of the program and the timing of receiving the message was not considered. In this evaluation, weight of constraints are $\{1, \cdots, 10\}$ and $d = 3$. A typical problem was picked for various number of nodes respectively. Results of 10 trials are averaged. The environment in which the experiment was performed is as follows: Intel Xeon 1.8GHz (HyperThreading), 512MB memory, 100Mbps Ethernet LAN, Windows2000, and MPICH1.2.5. Average of maximum number of cycles to find optimal solution in case of 4 processors is shown in Fig. 6. The result shows performance similar to the performance of the simulation. Average time to find the optimal solution is shown in Table. 1. Speedup rates show no significant effect on parallelism between algorithms .

6 Conclusion

Efficient methods for adopt algorithm are presented. The results shows the efficiency of the methods. Cycles and time to find optimal solution is reduced. Increase in the number of messages are relatively small. No significant effect on parallelism is shown. Improvement of method for learning solutions and applying to practical problems will be included in our future work.

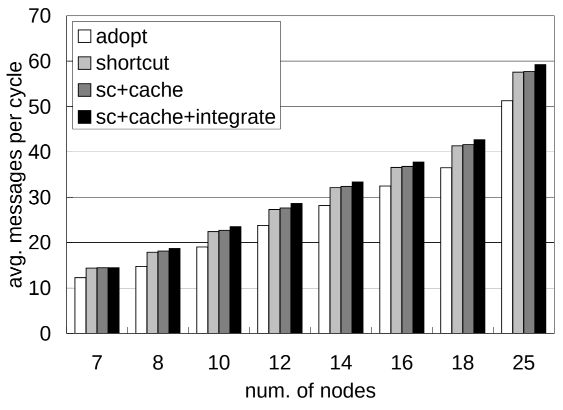

Figure 5. Average number of messages per cycle (weight of constraint=1,d=3)

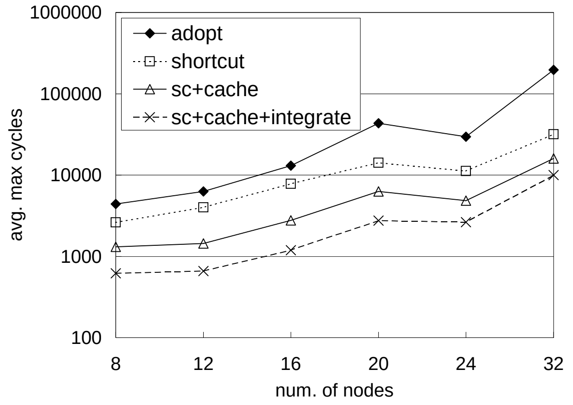

Figure 6. Average number of max cycles to find the optimal solution (MPI version, weight of constraint=$\{1, \cdots, 10\}$, d=3, 4 processors)

num. of nodes	method	num. of proc. 1	num. of proc. 4	speed up rate
8	adopt	2669	1823	1.5
	shortcut	3013	1748	1.7
	sc+cache	914	679	1.3
	sc+cache+int.	597	334	1.8
12	adopt	9576	5145	1.9
	shortcut	12250	5327	2.3
	sc+cache	3711	1750	2.1
	sc+cache+int.	2006	822	2.4
16	adopt	20831	9992	2.1
	shortcut	21673	9075	2.4
	sc+cache	6341	3070	2.1
	sc+cache+int.	3099	1377	2.3
20	adopt	124353	56539	2.2
	shortcut	79581	30174	2.6
	sc+cache	30813	12755	2.4
	sc+cache+int.	16195	6026	2.7
24	adopt	83953	37452	2.2
	shortcut	59130	22222	2.7
	sc+lru	21444	9009	2.4
	sc+lru+int.	13164	4845	2.7
32	adopt	855936	341583	2.5
	shortcut	277039	98245	2.8
	sc+cache	122330	45397	2.7
	sc+cache+int.	85022	28613	3.0

Table 1. Average time to find the optimal solution[ms] (MPI version, weight of constraint=$\{1, \cdots, 10\}$, d=3, 1 and 4 processor(s))

Acknowledgement

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (C), 15500037, 2003.

References

[1] T. Schiex, H. Fargier and G. Verfaillie, Valued constraint satisfaction problems: Hard and easy problems, *Proc. Int. Joint Conf. on Artificial Intelligence*, Montreal, Canada, 1, 1995, 631-639.

[2] E.C. Freuder and R.J. Wallace, Partial Constraint Satisfaction, *Artif. Intell.*, 58(1-3), 1992, 21-70.

[3] M. Yokoo, E.H. Durfee, T. Ishida and K. Kuwabara, The Distributed Constraint Satisfaction Problem: Formalization and Algorithms, *IEEE Trans. Knowledge and Data Engineering*, 10(5), 1998, 673-685.

[4] M. Lemaître and G. Verfaillie, An Incomplete Method for Solving Distributed Valued Constraint Satisfaction Problems, *In Proc. AAAI97 Workshop on Constraints and Agents*, Providence, USA, 1997.

[5] K. Hirayama and M. Yokoo, Distributed Partial Constraint Satisfaction Problem, *Proc. 3rd Int. Conf. on Principles and Practice of Constraint Programming* Linz, Austria, 1997, 222-236,

[6] J. Liu and K. P. Sycara, Exploiting problem structure for distributed constraint optimization, *Proc. 1st Int. Conf. on Multiagent Systems*, San Francisco, USA, 1995, 246-253.

[7] P.J. Modi, W. Shen, M. Tambe and M. Yokoo, An Asynchronous Complete Method for Distributed Constraint Optimization, *Proc. Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, 2003, 161-168.

[8] K. Hirayama and M. Yokoo, The Effect of Nogood Learning in Distributed Constraint Satisfaction, *Proc. 20th Int. Conf. on Distributed Computing Systems*, Taipei, Taiwan, 2000, 169-177.

[9] T. Schiex and G. Verfaillie, Nogood Recording for Static and Dynamic Constraint Satisfaction Problems, *International Journal of Artificial Intelligence Tools*, 3(2), 1994, 187-207.

[10] M. Yokoo, Asynchronous Weak-commitment Search for Solving Distributed Constraint Satisfaction Problems, *Proc. 1st Int. Conf. on Principles and Practice of Constraint Programming*, Cassis, France, 1995, 88-102.