Adaptive Quality Equalizing: High-Performance Load Balancing for Parallel Branch-and-Bound Across Applications and Computing Systems[*]

NIHAR R. MAHAPATRA AND SHANTANU DUTT

mahapatr@eng.buffalo.edu
Dept. of Electrical & Computer Eng.
State University of New York at Buffalo
Buffalo, NY 14260

dutt@eecs.uic.edu
Dept. of Electrical Eng. & Comp. Sc.
University of Illinois at Chicago
Chicago, IL 60607

Abstract

In this paper, we present an adaptive version of our previously proposed quality equalizing (QE) load balancing strategy that attempts to maximize the performance of parallel branch-and-bound (B&B) by adapting to application and target computing system characteristics. Adaptive QE (AQE) incorporates the following salient adaptive features: (1) Anticipatory quantitative and qualitative load balancing mechanisms. (2) Regulation of load information exchange overhead. (3) Deterministic load balancing in extended neighborhoods instead of just immediate neighborhoods as in non-adaptive QE. (4) Randomized global load balancing to fetch work from outside the extended neighborhood. AQE yields speedup improvements of up to 80%, and 15% on the average, compared to that provided by QE for several real-world mixed-integer programming (MIP) problems, and near-ideal speedups for two of the largest problems in the MIPLIB benchmark suite on an IBM SP2 system.

1. Introduction and Background

Branch-and-bound (B&B) is among the most popular global optimization methods used to solve NP-hard combinatorial optimization problems (COPs) [7]. Many practical search problems solved by B&B are sequentially intractable, consequently, parallel processing has been employed as an effective means to meet the computational requirements of such hard application problems. Sample applications include mixed-integer programming (MIP), airline crew scheduling, VLSI CAD, database design, and genetics [7]. In this section, we briefly review sequential and parallel B&B to provide the background and motivation for the subject of this paper, the adaptive quality equalizing load balancing strategy, presented in the remaining sections.

[*]N.R. Mahapatra was supported by startup funds from the State University of New York at Buffalo and S. Dutt was supported by NSF grant MIP-9210049. Cornell Theory Center provided access to its IBM SP2.

1.1. Sequential Branch-and-Bound

B&B performs a best-first search (BFS) for a least-cost leaf node (representing an optimal solution) starting from a single $root$ node corresponding to the COP $\mathcal{P}$ to be solved in a state-space tree $\mathcal{T}$ [8].[1] Each node in $\mathcal{T}$ represents a subproblem derived from $\mathcal{P}$ and has a cost that is a lower bound on the cost of an optimal solution to that subproblem. The set of unexplored nodes is stored in an *OPEN* list. We denote by $best\_soln$ the cost of the current best solution at any time during B&B's operation. Due to its BFS ordering, B&B explores only nodes with cost less than the optimal solution cost—these nodes are called *essential* nodes since they must be expanded by any search algorithm that guarantees optimality; all other nodes are termed *non-essential*.

1.2. Parallelization of B&B

B&B is most commonly parallelized on distributed-memory machines by first exploring the search space from the $root$ node to generate a starting node for each of the processors, and then conducting sequential BFS on each of the processors from its starting node. Processors broadcast any improvements in $best\_soln$ which is maintained consistent across all processors.

1.2.1. Best-Node Rank and Degree of Load Balance

We now define a few terms used in the sequel. We denote the set consisting of processor i and its neighbors by $neighbors(i)$. For a set $S \supseteq i$ of processors, the S-*rank* of a node u, denoted *rank*$_S(u)$, in processor i is the position of the first equal-cost node v in a non-decreasing cost ordering of nodes in the *OPEN* lists of those processors. The S-rank is referred to as u's *local rank* when $S = \{i\}$, as u's *neighborhood rank* when $S = neighbors(i)$, and as u's *global rank* when S is the set of all P processors (see Fig. 1 for an

[1]Other search orderings such as depth-first search can also be used in B&B, but in this paper we consider B&B with BFS.

illustrative example). Note that sequential B&B using BFS expands nodes in order of their (global) ranks, and consequently expands the minimum number of nodes. Therefore, in a parallel B&B algorithm, the S-rank of the best node in any processor $i \in S$ (i.e., rank$_S(best\_node_i)$) accurately reflects the degree of load balance within that set—the smaller it is, the better is the load balance, and vice versa. Ideally, rank$_S(best\_node_i) \leq |S|$, or when S is the set of all P processors, rank$_S(best\_node_i) \leq P$.

The above parallel B&B algorithm can be very scalable provided good load balancing methods are employed to address its following two inefficiencies. (1) *Starvation*: This occurs when processors run out of work/nodes and idle. (2) *Non-essential work*: This occurs when processors processing nodes in non-global-best-first order expand non-essential nodes. We have developed one such method called *Quality Equalizing* (QE) strategy comprising quantitative and qualitative load balancing schemes; these schemes are summarized next.

1.2.2. QE: Quantitative and Qualitative Load Balancing

Quantitative load balancing is performed by having processors that are expected to become idle soon—processors in which the number of nodes is decreasing and is below a certain threshold called *acceptor threshold*—request work from neighboring processors with the largest number of nodes. This mitigates the idling that would otherwise occur due to interprocessor communication latency. The amount of work transferred depends upon work load conditions in the immediate neighborhoods of the donor and acceptor processors [2].

The basic idea behind the qualitative load balancing scheme is to ensure that for any two neighboring processors i and j, the cost of i's best node $\leq$ the cost of j's sth best node or *threshold node*, i.e., to ensure that rank$_{\{i,j\}}(best\_node_i)$ $\leq$ rank$_{\{i,j\}}(threshold\_node_j)$; here $s \leq 2$ is the *span* and is a parameter that is inversely related to the degree of load balance and directly related to the overhead of load balancing. For this purpose, each processor reports its *threshold cost* (the cost of its threshold node) whenever it changes to its neighbors. We have derived the following results for QE.

Theorem 1 [3] *In the QE strategy:*

1. *The neighborhood rank of the best node in any processor can become at most $d \cdot (s-1)+1$ (i.e., a small constant factor of s more than optimal) before work transfer is triggered to reduce it, where s is the span and d is the degree or number of neighbors of a processor in the architecture.*
2. *If a processor has no essential nodes, and has at least one neighbor with s or more essential nodes, it will request essential nodes from such a neighbor.*

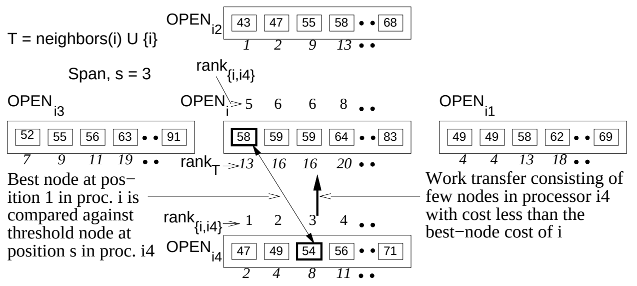

Figure 1: Processor i requests work from $i4$ when its best-node rank relative to $\{i, i4\}$ deteriorates beyond the threshold value s.

Since $d+1$ is the ideal worst-case neighborhood rank for the best node, for small s, parallel search using the QE strategy is close to an ideal near-neighbor scheme. Furthermore, through interaction between neighborhoods, good neighborhood load balance translates to good global load balance.

1.3. Randomized Load Balancing - Random Seeking

Here we describe the basic idea behind a randomized global load balancing scheme called *random seeking* (RS) that attempts to achieve "peer-peer" relationship between all processor pairs by probabilistically locating "source-sink" pairs of processors, and transferring work from the sources to the corresponding sinks [6]. Two processors i and j are defined to have a *source-sink* relationship if the threshold cost of i is less than the lead cost of j—note that near-neighbor QE (described in Section 1.2.2) performs work transfer between neighboring processors that have this relationship. They have a *sink-source* relationship if the threshold cost of j is less than the lead cost of i. Otherwise, i and j are defined to be *peers*. A source processor locates a relative sink by flinging probe messages to random processors. The probe messages, during their visits to processors, also collect load distribution information that is used by the initiating processor to efficiently regulate load-balancing activities. We have derived the following result which applies some time after parallel B&B has started when processors have obtained sufficient load distribution information.

Theorem 2 [6] *The average number of flings required by the random seeking scheme to transfer a node from a source to a sink processor is $\frac{1}{t}$, where t is the fraction of sink processors.*

From Theorem 2, RS is a negative feedback scheme—more the load imbalance (as reflected by the fraction of sinks), quicker the reaction (as reflected by the number of flings) to correct it. RS is thus a sophisticated, informed randomized scheme for performing fast qualitative load balance globally with very low overhead.

We have extensively evaluated, both analytically and empirically, the scalability and performance of our QE and

RS load balancing methods and shown that they are significantly better than those of previous deterministic and randomized methods [2, 3, 5, 6, 7]. Our analysis of load balancing methods revealed that the overheads of parallel B&B (processor idling, nonessential work, and load-balancing overheads) depend upon application *granularity* or node-expansion time and the topological characteristics of the target computing system, such as its degree and diameter [3, 7]. A load balancing strategy that minimizes idling and nonessential work with the least load balancing overhead, will have the best performance. Clearly, such a load balancing strategy must adapt to application and target system characteristics to extract the best performance. In the next five sections, we describe in detail our new adaptive load balancing method that is based on the near-neighbor, non-adaptive QE strategy discussed above and that also incorporates the RS scheme.

2. The Adaptive Quality Equalizing Strategy—An Overview

In this section, we present an overview of our adaptive quality equalizing (AQE) strategy. AQE adapts load balancing parameters (such as acceptor threshold and span) to application granularity and target system topology. While the topology is known *a priori*, granularity is known only at run time and may change, usually becoming smaller [7], as the parallel B&B search progresses. Therefore AQE adjusts parameter values continuously to current granularity. Parameters like acceptor threshold are local parameters and hence are adjusted independently in processors. Other parameters such as span are global parameters and should be consistent in all processors. Global parameters are updated in phases. A new phase is initiated by processor 0 after a minimum fractional change in average node expansion time (granularity) has occurred, after it has expanded a minimum number of additional nodes, and a minimum time interval has elapsed—all three of these conditions must be met. To initiate a new phase, processor 0 computes the new values of global parameters and broadcasts them to all other processors, which then update their parameter values.

The "average" node expansion time used to determine parameter values is an exponential average, denoted by t_e', computed as follows. Let $t_e'(n)$ denote the value of t_e' after the nth node expansion (i.e., the value of t_e' after its nth update). Let r_{n+1} denote the $(n+1)$th node expansion time. Then after the $(n+1)$th node expansion, we incorporate the new information, r_{n+1}, in the new average as [9]:

$$t_e'(n+1) = \alpha \cdot r_{n+1} + (1-\alpha) \cdot t_e'(n), \quad 0 \leq \alpha \leq 1.$$

The parameter α controls the relative weight of recent (1st term in the above Eq.) and past (2nd term in the above Eq.) history in the average estimation. We chose $\alpha = 0.25$ in our implementation so that recent history is captured well in the estimation. Consequently, t_e' reflects the average time to expand a node in the recent past well. Henceforth, whenever we refer to exponential averages, we mean computation similar to the above equation with $\alpha = 0.25$.

In the following sections, we describe in turn the various features of AQE.

3. Anticipatory Quantitative Load Balancing

In QE, acceptor threshold, which determines the amount of anticipation to hide the latency between work request and work procurement, is kept constant. Since this latency depends upon both application granularity and interprocessor message communication time, we make acceptor threshold adaptive in AQE. In our implementation, we keep track of an exponential average t_r' of the latencies to obtain work for all past work requests and an exponential average t_e' of the times to expand nodes, as stated earlier. Since the number of nodes decreases by one in at most t_e' time, acceptor threshold can be set to t_r'/t_e' to prevent idling. We update and set it to $1.2t_r'/t_e'$ after every work procurement—the factor of 1.2 is used to make the anticipation conservative.

4. Anticipatory Qualitative Load Balancing

Due to latency between work request and procurement, non-essential work may be performed in the QE strategy by a processor i even if essential nodes were available in a neighboring processor at the time of the work request from i. To address this latency problem, we use a general anticipatory mechanism in AQE of which QE is a special case with zero amount of anticipation. We call the node at position λ the *lead node* and the node at position τ the *threshold node*, where $\tau > \lambda \geq 1$. *Span* is then defined as $s = \tau - \lambda + 1 \geq 2$. In QE, $\lambda = 1$ and $s = \tau$. The basic idea is to check for any cost deterioration in the *lead node* (instead of the $best\_node$ as was done without the anticipatory mechanism) of a processor i relative to the *threshold node* of any neighbor j, and to trigger work transfer to correct this condition just before the lead node becomes the $best\_node$ in processor i at a later time. We have derived expressions in [7] for λ and τ in terms of granularity and message latency. We have also determined in [7] that span s should be increased at a rate slightly less than d, the number of neighbors of a processor; when qualitative load balancing is performed in an extended neighborhood (see Section 6.1), d should be replaced by the number of extended neighbors. We vary global parameters λ, τ, and span in phases as discussed earlier.

5. Regulating Qualitative Load Information Exchange Overhead

For small granularity applications, the qualitative load balancing overhead per node expansion can be high. The overhead due to work transfer can be reduced by increasing span. This, however, may not reduce the overhead of load-information exchange, since threshold cost, the cost c_τ of the node at position τ, is reported whenever it changes. To circumvent this problem, instead of just a threshold cost, we use a *threshold cost interval*, $\delta \geq 1$, defined as follows. We define $\tau_1 = \max(\tau - \delta/2, \lambda + 1)$ and $\tau_2 = \tau_1 + \delta - 1$, and c_{τ_1} and c_{τ_2} as the costs of the nodes at positions τ_1 and τ_2, respectively; $\delta = 1$ in QE. Instead of processors reporting c_τ whenever it changes, they now report c_{τ_2} whenever either the current c_{τ_1} exceeds the c_{τ_2} at the time of the last load-information report (i.e., when the current cost interval $[c_{\tau_1}, c_{\tau_2}]$ stops overlapping the previous $[c_{\tau_1}, c_{\tau_2}]$), or the current c_{τ_2} is smaller than the c_{τ_2} previously reported, a less likely event. Note that in QE, at any given time, there are at least τ nodes in a processor of cost no more than the threshold cost (c_τ) last reported, while in AQE using threshold cost interval, at any given time, there are at least τ_1 nodes of cost no more than the c_{τ_2} last reported. Qualitative load-information exchange overhead can be reduced by increasing δ, therefore, we vary δ inversely with granularity.

6. Combining Deterministic and Randomized Load Balancing

Ideally, we would like to have deterministic global load balance. However, in practice, the costs of ensuring load balance deterministically over the entire processor space will be prohibitive. Therefore we can perform deterministic load balance in small processor neighborhoods, the size of the neighborhood being dependent upon the application granularity, and randomized load balance beyond the neighborhoods. These are described below.

6.1. Deterministic r-Neighborhood Load Balancing

Following our terminology in Sec. 1.2.1, let $neighbors^r(i)$, for $r \geq 1$, denote all processors at a distance of r or less from processor i (including processor i); this is also called the *r-neighborhood* of i. We can extend the near-neighbor QE strategy of Section 1.2.2 by having processors perform load balancing with respect to all processors in their r-neighborhoods instead of only their near-neighbors. The following result, similar to the one derived for the near-neighbor QE strategy, is easily obtained.

Theorem 3 *In AQE with r-neighborhood load balancing:*

1. *The k-neighborhood rank of the best node in any processor $\leq d_k(s - 1) + 1, 1 \leq k \leq r$, otherwise work transfer will be triggered to reduce it, where s is the span and d_k the number of processors in a k-neighborhood.*
2. *If a processor has no essential nodes, and has at least one processor in its r-neighborhood with s or more essential nodes, it will request essential nodes from such a processor.*

Thus the above scheme will provide good r-neighborhood load balancing. In our implementation, we perform both quantitative and qualitative r-neighborhood load balancing. The choice of r is related to the choice of span, since as noted in Section 4, span is increased at a rate slightly less than d_r, the number of processors in an r neighborhood. Also, since a smaller span means more stringent load balancing and consequently more load-balancing overhead, we vary span inversely with application granularity.

6.2. Randomized Global Load Balancing

To complement the r-neighborhood deterministic load balance achieved by the above scheme, we use the random seeking scheme of Section 1.3 outside of the r-neighborhoods to achieve probabilistic global load balance. For this purpose, probe messages are flung to processors outside the r neighborhoods. Among processors that are more than a distance r apart, the interaction via the deterministic r-neighborhood scheme between processors that are farther apart will be less and slower compared to those that are nearer. Therefore we keep the probability of flinging probes to processors at a distance $l > r$ inversely proportional to l and directly proportional to the number of processors at that distance.

In RS, a certain number of probe messages (proportional to the number of relative sources and inversely proportional to the probability that a probe will locate such a source) is flung from sink processors in every fixed time interval Δ. This way probe-flinging overhead is automatically adapted to grain size—the number of probes flung is inversely related to granularity.

The RS scheme thus enables quick work transfer from source neighborhoods to sink neighborhoods, after which the deterministic r-neighborhood load balancing scheme facilitates distribution of the work received within the sink neighborhood.

7. Performance Results

In this section, we give performance results for adaptive and non-adaptive QE by solving several real-world MIP problems from the MIPLIB benchmark suite [1], drawn from a variety of applications, on an IBM SP2. Table 1 shows improvements in speedup (the ratio of sequential execution time T_1 to parallel execution time T_P on P processors) obtained by using AQE over QE for five problems on 16 processors using both a high-performance switch and Ethernet as interconnects. Observe that the performance with a high-performance switch is generally higher compared to that with an Ethernet, although the difference is not very great. This is likely due to the anticipatory nature of our load balancing schemes which effectively mitigate the impact of interconnect latencies. Although, there is one case (out of ten) where adaptive QE deteriorates performance to some marginal extent (5.35%), there are several instances where it yields considerably better performance, up to as much as almost 80%, and the average speedup improvement obtained is 15%. Next, Table 2 gives speedups obtained with

Problem	Cnstr.	Vars.	Int.	0/1	Nodes	T_{16} (ms)	Inteconn.	Spdup.	% ΔSpd.
bm23	20	27	27	27	631	2040	HPS	13.333	+20.34
bm23	20	27	27	27	559	1640	Ethernet	13.015	+2.04
misc01	54	83	82	82	697	2499	HPS	13.068	-0.70
misc01	54	83	82	82	701	2350	Ethernet	12.575	+8.62
mod013	62	96	48	48	411	1380	HPS	12.580	+29.16
mod013	62	96	48	48	399	1410	Ethernet	10.417	-5.35
pipex	24	48	48	48	2377	5789	HPS	14.571	+0.28
pipex	24	48	48	48	2577	6349	Ethernet	14.745	+8.45
stein15	36	15	15	15	275	710	HPS	10.794	+79.96
stein15	36	15	15	15	283	770	Ethernet	11.616	+6.97

Table 1: Comparison of speedups obtained by the adaptive and non-adaptive QE strategies for five problems from the MIPLIB benchmark suite on 16 processors of an IBM SP2. Columns 1 through 11 of the table give the following information: (1) problem name, (2) number of constraints, (3) number of variables, (4) number of integer variables, (5) number of integer variables that are binary, (6) number of essential nodes, (7) execution time on 16 processors, (8) interconnect type, (9) speedup obtained with adaptive QE, (10) speedup improvement obtained with adaptive QE over non-adaptive QE speedup.

Problem	Cnstr.	Vars.	Int.	0/1	Nodes	T_8 (ms)	Inteconn.	Spdup.
bell3a	123	133	71	39	48523	273674	HPS	7.832
rentacar	6803	9557	55	55	2820	3242789	Ethernet	7.454

Table 2: Speedups obtained using adaptive QE strategy for two problems from the MIPLIB benchmark suite, one with the maximum number of non-binary integer variables (*bell3a*) and the other with the maximum number of constraints (*rentacar*).

AQE on eight processors with a high-performance switch interconnect for two large MIPLIB problems: `bell3a`, the problem in MIPLIB with the most number of general or non-binary integer variables, and `rentacar`, the problem in MIPLIB with the most number of constraints. The speedups obtained are almost ideal.

8. Conclusions

In this paper, we presented an adaptive version of our near-neighbor QE strategy that incorporates the following adaptive features: anticipatory quantitative and qualitative load balancing mechanisms, regulation of load-information exchange overhead, and integration of deterministic and randomized load balancing methods. We explained how these features are made adaptive with respect to application granularity and target system characteristics (such as degree, diameter, and communication delay) to maximize performance. We obtained significant speedup improvements of up to 80%, 15% on the average, using AQE over QE for several real-world MIP problems from the MIPLIB benchmark suite, drawn from a variety of applications, on an IBM SP2 with Ethernet and high performance switch as interconnects. Further, we obtained near-ideal speedups for two of the largest problems in MIPLIB. To the best of our knowledge, this is the first time that an adaptive load balancing strategy that adapts to both application and target-system characteristics with such generality has been presented. In the future, we will augment AQE and test it on more application problems like airline crew scheduling, and on other parallel and distributed platforms with larger number of processors.

References

[1] R.E. Bixby, E.A. Boyd, and R.R. Indovina, "MIPLIB: A test set of real-world mixed-integer programming problems," *SIAM News*, Vol.25, No.2, pp.16, 1992.

[2] S. Dutt and N.R. Mahapatra, "Parallel A* algorithms and their performance on hypercube multiprocessors," *Seventh Int'l Par. Proc. Symp.*, pp.797-803, Apr. 1993.

[3] S. Dutt and N.R. Mahapatra, "Scalable load balancing strategies for parallel A* algorithms," *J. of Par. and Distr. Comp.*, Vol.22, No.3, pp.488-505, Sep. 1994.

[4] N.R. Mahapatra and S. Dutt, "Scalable duplicate pruning strategies for parallel A* graph search," *5th IEEE Symp. Par. and Distr. Procg.*, pp.290-297, Dec. 1993.

[5] N.R. Mahapatra and S. Dutt, "New anticipatory load balancing strategies for parallel A* algorithms," *DIMACS Series in Disc. Math. and Theor. Comp. Sc.—Par. Procg. of Disc. Opt. Probs.*, Pardalos, et al. (Eds.), Vol. 22, pp. 197-232, 1995.

[6] N.R. Mahapatra and S. Dutt, "Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing," *Proc. 10th Int. Par. Procg. Symp.*, pp. 881-885, Honolulu, Hawaii, Apr. 1996.

[7] N.R. Mahapatra, "Scalable, high-performance parallel branch-and-bound algorithms for solving large combinatorial optimization problems," *Ph.D. Thesis*, Dept. of Elec. Eng., U. of Minnesota, Minneapolis, 1997.

[8] E. Rich, "Artificial Intelligence," *McGraw Hill*, New York, pp.78-84, 1983.

[9] A. Silberschatz and P.B. Galvin, *Addison-Wesley*, Reading, Massachusetts, pp.138-140, 1994.