Augmented Binary Hypercube: A New Architecture for Processor Management

Hari Lalgudi, Ian F. Akyildiz, *Fellow, IEEE*, and Sudhakar Yalamanchili, *Member, IEEE*

Abstract—Augmented Binary Hypercube (AH) architecture consists of the binary hypercube processor nodes (PNs) and a hierarchy of management nodes (MNs). Several distributed algorithms maintain subcube information at the MNs to realize fault tolerant, fragmentation free processor allocation and load balancing. For efficient implementation of AH, we map MNs onto PNs, define and prove infeasibility of ideal mappings. We propose easily implementable non-optimal mappings, having negligible overheads on performance. Extensive simulation studies and performance analysis conclude that these algorithms realize significantly better average job completion time and higher processor utilization, as compared to the best sequential allocation schemes and parallel implementation of Free List [7]. AH algorithms can be tuned or adapt to the job and system characteristics, and resource management traffic.

Index Terms—Augmented binary hypercube, update algorithms, ideal mapping, resource management traffic.

1 INTRODUCTION

THIS paper focuses on two resource management problems for the binary hypercube—

1) *processor allocation* (PA), and
2) *load balancing* (LB).

The processors are resources to be allocated to the set of tasks in a job. A subcube is allocated, instead of arbitrary processors, to efficiently manage resources and minimize communication overheads. However, this causes fragmentation and yields lower average processor utilization. Static or dynamic load balancing can increase processor utilization by reducing fragmentation. The topology makes it nontrivial to detect the availability of a subcube. Determining the maximum size available subcube is NP-complete [8]. Many of the first approaches utilized off-line serial computations for processor allocation/deallocation [2], [7], [3]. For scalable and reliable solutions to *PA* and *LB*, the algorithms must be parallel and on-line. Parallel implementations of of the above serial algorithms have been proposed in [2], [3], [7], but require dedicated processors.

Our approach is to logically augment the binary hypercube with *Management Nodes (MNs)*. Each MN contains the corresponding subcube status information. Links are introduced between *MN*s and processor nodes *(PNs)* forming a ternary hypercube topology. A class of fault tolerant algorithms search and update the *MN* status information. These algorithms realize distributed, fragmentation-free, fault tolerant, processor allocation and load balancing. From a practical standpoint, the topology is realized by mapping *MN*s onto *PN*s.

We propose and evaluate several mapping functions in terms of search, update, completion times, and utilization.

• *H. Lalgudi is with Zeitnet Inc., 5150 Great America Parkway, Santa Clara, CA 95054. E-mail: hari.lalgudi@zeitnet.com.*
• *I.F. Akyildiz and S. Yalamanchili are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {ian, sudha}@ee.gatech.edu.*

Manuscript received Aug. 3, 1993; revised Oct. 29, 1995.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96069.

2 THE AUGMENTED BINARY HYPERCUBE

2.1 Definitions of Augmented Hypercubes

The generalized *r_ary* hypercube consists of r^m number of nodes, where each node v_i has an m-digit r-ary representation $i_{m-1} \ldots i_0$, where $i_p \in \{0, 1 \cdots, r-1\}$ for $0 \le p \le m-1$. A communication link exists between two nodes v_i and v_j if and only if the m digit representation of v_i and v_j differ only in one digit position. The resulting network is a regular graph with degree $(r-1)m$, node connectivity $(r-1)m$ and diameter m. The binary hypercube is a special case when $r = 2$.

DEFINITION 1 (AH). Augmented Binary Hypercube *consists of PNs and MNs. A node v_i has an m digit ternary representation $i_{m-1} \cdots i_0$, where $i_p \in \{0, 1, *\}$ for $0 \le p \le m-1$.*

DEFINITION 2 (PN). *A* Processor Node *v_i, represented by $i_{m-1} \cdots i_0$, is an AH node, with the constraint $i_p \in \{0, 1\}$ for $0 \le p \le m-1$. PN corresponds to a binary hypercube node.*

DEFINITION 3 (MN). *A* Management Node *v_i, of AH, is the* center *of the k* dimensional *subcube, S_{v_i}, of a binary hypercube. In the m digit representation of v_i, ($i_p \in \{0, 1, *\}$, $0 \le p \le m-1$), k of the digits $i_{j_1}, \ldots, i_{j_m} \in \{*\}$ and the remaining $(m-k)$ digits $\in \{0, 1\}$. Subcube S_{v_i} is spanned by the dimensions $j_1, \cdots, j_m$.*

DEFINITION 4 (*d*). $d(v_i) = \sum_{k=0}^{m-1} g_k$, *where $g_k = 1$ iff $i_k \in \{*\}$, and $g_k = 0$ otherwise.*

DEFINITION 5 (center). *The vertex v_j is called the* center *of the subcube, S_{v_i} of dimension $d(v_j)$, along the dimensions k such that $j_k \in \{*\}$.*

All links in the binary hypercube are included in *AH*. Links between the *MN*s and between *MN*s and *PN*s can be added in either of the following two ways.

CASE (a) **AH1**. Two nodes v_i and and v_j, with representation $i_{m-1} \cdots i_0$ and $j_{m-1} \cdots j_0$, have a link if $\exists k$, $i_k = *$ and $j_k \in \{0, 1\}$ and for all l, $0 \le l \le \{m-1\}$ and $l \ne k$, $i_l = j_l$.

CASE (b) **AH2**. Two nodes v_i and v_j have a link if the m digit representation of v_i and v_j differ only in one position.

In Case(a), v_i is connected to v_j, if and only if, $S_{v_j} \in S_{v_i}$ and $d(v_i) = d(v_j) + 1$. In Case(b), v_i is connected to v_j, if their m-digit ternary representations differ in one position.

The properties of AH1 and AH2 is stated in [9], and is summarized as follows. Both AH1 and AH2 have 3^m nodes, 2^m PNs and $(3^m - 2^m)$ MNs. AH1 has $m(3^{m-1} + 2^{m-1})$ links, degree of $2m$ for PN and $(m + k)$ for a MN at the center of a k dimensional subcube, and diameter of $3m/2$. The properties of AH2 follow from the ternary hypercube of the same dimension, namely $m3^m$ links, degree m and diameter m. For *PA*, the communication cost in *AH*1 is not affected due to increased diameter. *AH*1 has k independent paths of length k, and $(m-k)$ paths of length $(k+3)$, between a *MN* v_i and *PN* v_j, where $d(v_i) = k$ and $v_j \in S_{v_i}$. *AH*2 has m node disjoint paths between any two nodes. Only k paths are used for information update. Additional *AH*2 links allow fault tolerant routing.

By definition, there is an *MN* v_i for every possible subcube S_{v_i}. The status of S_{v_i} and its constituent *PN*s is maintained at *MN* v_i, and is obtained by exchanging information with other *MN*s con-

Strategy	# Subcubes	Allocation	Deallocation	Memory	Type	Flexibility
Buddy	$2^{n+1}-1$	$O(2^n)$	$\Theta(2^m)$	$\Theta(2^n)$	first–fit	No
GC	3.2^n-3	$O(2^n)$	$\Theta(2^m)$	$\Theta(2^n)$	first–fit	No
Bin. Tree	3.2^n-3	O(n)	O(n)	$O(2^n)$	best–fit	No
Mod. Buddy	$n.2^n+1$	$O(n.2^n)$	$\Theta(2^m)$	$\Theta(2^n)$	first–fit	No
Mult. GC	3^n	$O(C^n_{\lfloor \frac{n}{2} \rfloor}.2^n)$	$O(C^n_{\lfloor \frac{n}{2} \rfloor}.2^m)$	$O(C^n_{\lfloor \frac{n}{2} \rfloor}.2^n)$	first–fit	No
MSS	3^n	$O(2^{3^n})$	$O(n.2^n)$	$O(n.3^{2n})$	best–fit	No
PC Graph	3^n	$O(n^{-2}3^{3^n})$	$O(n^{-2}3^{2n})$	$O(n^{-1}3^{2n})$	best–fit	No
FL	3^n	$O(n^2)$	$O(n^2\, 2^{2n-2m})$	$O(n\, 2^n)$	best–fit	No
AH	3^n	$O(\frac{n!}{u!}+k\mu u)$	$O(k\, \mu\, u)$	$O(3^n)$	best–fit	Yes

Fig. 1. Comparison of various strategies.

tained in S_{v_i} it. Any *MN* can be viewed as the root of a hierarchy of *MN*s with *PN*s at the leaves. The ratio of *PN*s to the total nodes (*NN*), $(2/3)^m$, shows that with increasing m, the number of *MN*s, links and the degree of each node increases rapidly. Secondly, the same distributed algorithms should be implementable on existing hypercubes. Hence we do not realize *MN*s as distinct physical nodes, but map the *MN*s onto *PN*s.

2.2 Mapping of AH Nodes onto Binary Hypercube Nodes

DEFINITION 6 (Γ). *The function* $\Gamma : \Re_1 \mapsto \Re_2$ *(where* $\Re_1 = \{0, \cdots, 3^m\}$ *and* $\Re_2 = \{0, \cdots, 2^m\}$*) maps the nodes of the AH1 or AH2 to the PNs of AH1 or AH2.*

This mapping implies decreased parallelism in update algorithms (Section 3) and increased overheads on the *PN*s due to *MN* functions. An *ideal* mapping function, reducing these overheads, is defined as follows:

DEFINITION 7 (Γ_i). *Ideal mapping function* Γ_i *is such that, if* $\Gamma_i(v_1) = m_1$, $\Gamma_i(v_2) = m_2$,

1) ***Parallelism Constraint:*** *If* $d(v_1) = d(v_2)$, *then* $m_1 \neq m_2$.
2) ***Dilation Constraint:*** *If* $d(v_1) = d(v_2) + 1$ *and there exists a link between* v_1 *and* v_2, *then there is a link between* m_1 *and* m_2.

The parallelism constraint ensures that there is no loss of parallelism by mapping *MN*s of same level on different *PN*s, although there is a loss of bandwidth. The dilation constraint ensures that adjacency of nodes in *AH* is preserved and hence, messages traverse the same number of physical links in *AH* and mapped *AH*.

THEOREM 1. *An ideal mapping function,* Γ_i, *does not exist.*

PROOF. Refer to [9]. □

Hence, we propose *nonideal* mapping schemes, preserving dilation, not parallelism constraint. This results in decreased parallelism but preserves message delays. The mapping and inverse mapping is implemented with a mask table for each *PN*, with fields as shown in Fig. 2c. On receiving a message, the *PN* searches the mask table for the *MN*, to which the message is addressed. It completes the requisite action and updates the *MN* entry.

Let $\Gamma(v_i) = M_j$, where the m digit representation of v_i is $i_{m-1} \cdots i_0$, and that of M_j is $j_{m-1} \cdots j_0$, and $i_k \in \{0, 1, *\}$ and $j_k \in \{0, 1\}$ ($k = 0, \cdots, m-1$). The mapping strategies are:

1) *Mask Mapping:* If $i_k \in \{0, 1\}$, then $j_k = 0$, and if $i_k = *$, then $j_k = 1$.
2) *Star Mask Mapping:* If $i_k \in \{0, 1\}$, then $j_k = i_k$, and if $i_k = *$, then $j_k = 1$.
3) *Run Mask Mapping:* Let the dimension of $v_i = d(v_i)$.

If $(\forall l\ (l \leq k), i_l \in \{0, 1\})$, if $(i_k \in \{0, 1\})$, then $j_k = i_k$, and if $(i_k = *)$, then $j_k = 1$.
Else if $(\forall l\ (l \geq k)), i_l \in \{0, 1\}$, then if $(i_k \in \{0, 1\})$, then $j_k = i_k$, and if $(i_k = *)$, then $j_k = 1$.
Else if $(\exists q, \forall l\ (l \leq q), i_l \in \{0, 1\})$ and if$(k > (q + d(v_i)))$, then $j_k = 1$, else $j_k = 0$.

In *Mask Mapping*, the *MN*s at the same level map to different *PN*s. However, a large number of small k dimensional nodes (2^{m-k}), map to the same node. In *Star Mask Mapping*, these (2^{m-k}) nodes map separately onto different nodes, lowering mapping overheads. However, two *MN*s, of same dimension, having * and 1 in the same position map to the same *PN*, reducing parallelism. In *Run Mask Mapping*, all digits, between the first and last digit having * (*run* of digits), are masked to 1. The other digits retain the same values. This strategy violates the dilation constraint but it reduces the overheads on parallelism. The mapping distribution can be found in [9].

3 INFORMATION UPDATING IN THE AUGMENTED ARCHITECTURE

We propose efficient distributed algorithms for maintaining subcube information in AH for *PA* and *LB*. *MN* v_i maintains S_{v_i} specific information. For *PA*, it is the number of available *PN*s in S_{v_i}, *avail*(v_i). For *LB*, it is the load on S_{v_i}, *load*(v_i). For PN v_i, *avail*(v_i) = 1 if v_i is available, 0 otherwise. For MN v_i, $avail(v_i) = \sum_{j=1}^{2^m} avail(v_j)$, where v_j is a *PN*, $v_j \in S_{v_i}$, and *avail*(v_j) = 1. For MN v_i, $load(v_i) = \sum_{j=1}^{2^m} load(v_j)$, where v_j is a *PN*, $v_j \in S_{v_i}$. The following phases search and update this information:

Phase 1 (Search Phase): *PA* searches for the optimal subcube (first or best fit) with available nodes for allocation. *LB* searches for the most heavily and lightly loaded subcubes.
Phase 2 (Updating Stage 1): On allocation of tasks, *PA* updates subcube information. *LB* updates load information on the *PN*s and *MN*s on allocation or relocation of tasks.
Phase 3 (Updating Stage 2): Subcube information must be updated in *MN*s for deallocated processors or those freed by task relocation.

Fig. 2a shows the Phase 1 template and Fig. 2b that of Phases 2/3. The search phase begins at H_n, MN at the center of the hypercube. H_n sends a *search* message with m and k ($k < 2^m$). The update phase begins at the allocated/deallocated *PN*. For consistency, *either* a search phase *or* multiple update phases can be in progress at any time. We propose algorithms which optimize costs, in terms of the number of messages sent.

```
If (recvdMsg == {'search', v_i, v_p, m, k}) {
   /* v_p = parent node(sender), m = reqd. subcube size,
   k = appln. specific parameter */
   If (d(v_i) == m) /* C1 */
      {set replyMsg based on info. at v_i, k}  /* S1 */
   Else If (d(v_i) > m)
      {Either set replyMsg based on info. at v_i}   /* S2 */
      {Or ∀v_j s.t. d(v_j) = d(v_i) − 1 and S_vj ⊂ S_vi
         sendMsg(v_j, {'search', v_j, v_i, m, k});
         recvReply(reply);
         {set replyMsg based on reply} /* S3 */
      }
   Else /* d(v_i) < m */
      {Either Do nothing (algorithm specific choice)}
      {Or ∀v_j s.t. d(v_j) = d(v_i) − 1 and S_vj ⊂ S_vi
         sendMsg(v_j, {'search', v_j, v_i, m, k});
         recvReply(reply);
         {set replyMsg based on reply} /* S4 */
      }
   sendReply(v_p, replyMsg);
}
```

(a) Search phase at node v_i

```
Foreach allocated processor v_i,
   SendAllocatedMsg(v_j, {'update', v_j, v_i});
   /* where d(v_j) = 1, and v_i ∈ S_vj */
Foreach MN v_j
   If ' (recvdMsg == {'update', v_j, v_i}) {
      {Update info. at v_j} /* S5 */
      AC_vj = (AC_v + 1)mod(d(v_j));
      If (AC_vj == 0){
         {Update v_j info. at v_j} /* S6 */
         ∀v_c s.t. d(v_c) = d(v_j) + 1 & S_vj ⊂ S_vc
         SendAllocatedMsg(v_c, {'update', v_c, v_j});
      }/* End of If (Allocation Counter is 0) */
   }/* End of Updating Algo for MN v_j */
```

(b) Update phase for immediate update

Fields	Functions
Valid Bit	Allocated Mask Valid
Mask	The Actual Mask
State	MN State (e.g., counter, load)
Others	Application Specific Fields

(c) Mask table on each PN

Fig. 2. Search and update phases and mask table.

3.1 Algorithm 1: Immediate Update

MN v_i maintains consistent S_{v_i} information and is immediately informed of any state change of a $PN \in S_{v_i}$. This strategy is useful when search costs dominate update costs.

DEFINITION 8 (CN). *Nodes v_i and v_{i1} are* Complementary Nodes *(CN), if*

$$d(v_i) = d(v_{i1}) = d(v_j) - 1,\ S_{v_i} \subset S_{v_j},\ S_{v_{i1}} \subset S_{v_j} \text{ and } S_{v_i} \cap S_{v_{i1}} = \phi.$$

Phase 1: For *PA*, in Fig. 2, *k* is the number of processors to be allocated. When $avail(v_i) < k$, the processors are allocated in two disjoint smaller subcubes. *S*1 and *S*2 check if $avail(v_i) \geq k$. If so, *replyMsg* indicates success if v_i can be allocated, failure otherwise. Allocation begins at v_i, such that, $avail(v_i) \geq k$, $\forall v_j$, $d(v_j) = d(v_i) - 1$ and $S_{v_j} \subset S_{v_i}$, $avail(v_j) < k$. *MN* v_i sends allocate messages to two *CN*s v_j and v_{j1}, where $S_{v_j}, S_{v_{j1}} \subset S_{v_i}$, and $d(v_j) = d(v_{j1}) = d(v_i) - 1$. *S*2 also sends allocate messages to the *CN*s. *S*4 selects v_j and v_{j1} and receives the reply. Allocate messages terminate at *PN*s. Multiple messages cannot reach the *PN*s, as the strategy recursively allocates *PN*s in disjoint subcubes.

For *LB*, *k* indicates if the search is for a maximal or minimal loaded subcube. *S*1 sets *replyMsg* to $load(v_i)$. Based on *k*, *S*3 checks if the load in *reply* is greater than the maximal load or is less than the minimal load of the previously checked *m*-dimensional subcubes, and accordingly sets *replyMsg*.

Phases 2 and 3: The updating propagates from *PN* to H_n. *MN* v_j has $d(v_j)$ node disjoint paths of length $d(v_j)$, to PN v_i, which is allocated/deallocated. MN v_j has a modulo–$d(v_j)$ counter to count received update messages, and allocation/deallocation at v_j is complete on the receipt of $d(v_j)$ messages. The counter ensures the correctness of the distributed algorithm, even when multiple nodes in S_{v_i} are allocated simultaneously, AC_{v_j} is the *Allocation Counter* of node v_j. In Fig. 2b, for *PA*, *S*6 decreases $avail(v_j)$ in Phase 2 and increases $avail(v_j)$ in Phase 3. For *LB*, the message contains $load(v_i)$, which is incremented with each allocation. *S*5 increments $load(v_j)$ in Phase 2, and decrements $load(v_j)$ in Phase 3.

3.2 Algorithm 2 (Lazy Update)

MN v_i does not have consistent S_{v_i} information. In the search phase, it is collected *on demand* from the *PN*s. When update requests are high, the updating messages are not sent, reducing the updating costs at the expense of higher search costs.

Phase 1 (Search): The algorithm is the same as the Phase 1 of Section 3.1, except that the *C*1 in Fig. 2 is replaced by the following condition: {If ($visited(v_i)$)/* C2 */}.

For *PA* and *LB*, *S*1 in Fig. 2a is as follows: If ($d(v_i) = m$) {If ($avail(v_i) \geq k$), *replyMsg* indicates success with node value as v_i}. Else If ($d(v_i) < m$) replyMsg contains $avail(v_i)$. Else If ($d(v_i) > m$) {since v_i is already visited, *replyMsg* has previous search result. $avail(v_i)$ is set to *replyMsg*}.

*S*3 is as follows: If ($d(v_i) \geq m$) {Let v_{j1} and v_j be CNs. If $S_{v_{j1}}$ information is available in *replyMsg*, then $visited(v_i)$ is set and $avail(v_i) = avail(v_j) + avail(v_{j1})$ or $load(v_i) = load(v_j) + load(v_{j1})$}, else *replyMsg* is set to $avail(v_j)$ or $load(v_j)$, and search message sent to v_{j1}. If ($d(v_j) = m$ and $avail(v_j) \geq k$) *replyMsg* indicates *success* at v_i else *failure*}.

If ($d(v_i) < m$) {*replyMsg* contains $load(v_j)$ or $avail(v_j)$.} For *PA*, search terminates after v_j and v_{j1} information is obtained. For *LB*, depending on *k*, the search continues for all smaller dimensional subcubes to find minimally or maximally loaded subcubes

Phases 2 and 3: No information updating is done.

3.3 Algorithm 3 (Intermediate Update)

DEFINITION 9 (*u*). *Update height (u) is the size of the subcube, such that an MN v_i keeps consistent information, by constant updating during allocation/deallocation, iff $d(v_i) \leq u$.*

A node v_i executes **Immediate Update** if $d(v_i) \leq u$ and **Lazy Update** if $d(v_i) > u$. By choosing *u*, it allows trade-offs between the search and update costs.

3.4 Algorithm 4 (Adaptive Update)

DEFINITION 10 (wavefront). *The wavefront consists of nodes maintaining consistent information, where update and search messages, for collecting consistent information, terminate.*

A node v_i executes either the **Immediate** or the **Lazy Update**, depending on the fraction of the search to the total number of messages, $\alpha_{v_i}^{curr}$, maintained locally. Let $\alpha_{v_i}^{low}$ and $\alpha_{v_i}^{high}$, be the cut-off limits for v_i ($0 \le \alpha_{v_i}^{low} \le \alpha_{high}^{v_i}$). When $\alpha_{v_i}^{curr} < \alpha_{v_i}^{low}$, v_i extends the *wavefront* to the centers of the smaller dimensional subcubes, by sending messages to them to use *Lazy Update*, thus saving on update costs. When $\alpha_{v_i}^{curr} > \alpha_{v_i}^{high}$, v_i contracts the wavefront to the centers of the larger subcubes, by changing to *Immediate Update*, thus saving search costs. This strategy can adapt to unknown job characteristics.

The *PA* schemes in [2], [7], [3] involve identification of an m dimensional subcube such that $2^{m-1} \le k < 2^m$, causing fragmentation. Our distributed allocation strategies are *fragmentation free*, (similar to [1]), as $avail(v_i)$ allows

1) allocation in partially allocated subcubes of v_i, and
2) partitioning of tasks to v_j, when $avail(v_i) \ge k$, $avail(v_j) < k$, $S_{v_j} \in S_{v_i}$ and $d(v_j) \ge m$.

4 Performance Analysis of the Algorithms

Since static and dynamic *LB* use *PA*, we study only *PA*. We compare Buddy, Modified Buddy, Maximal Set of Subcubes (*MSS*) [5], Prime Cube (PC Graph) [11], Gray Code scheme (*GC*) [2], Tree Collapsing scheme (*TC*) [3], Free-List scheme (*FL*) [7] with *AH* in terms of time required for allocation, deallocation, space complexity and number of subcubes recognized (note $k \le 2^m$, the size of the subcube required). The cost in *AH* is the time spent in sending or receiving messages. Unlike centralized algorithms, the performance of *AH* is dependent on the message delay, which depends on request rates, cube sizes, traffic patterns, queuing at intermediate nodes, buffer sizes, routing algorithms etc. *GC*, *TC*, and *FL* are centralized whereas *AH* algorithms are distributed. Only *AH* takes into account the job and system characteristics, e.g., λ, η, u, μ.

4.1 Complexity Analysis

Lazy Update and *Immediate Update* are special cases of *Intermediate Update* algorithm, when $u = 0$ and m, respectively. *Adaptive Update* costs are bounded by the updating costs of *Immediate Update* and the search costs of *Lazy Update*. Hence, we only analyze the worst case costs (WC) for the *Intermediate Update* as follows: during Phase 1, messages can only reach level u. Any node v_i has to send to v_j, at most $d(v_i)$ messages (where $S_{v_j} \in S_{v_i}$, $d(v_j) = d(v_i) - 1$), leading to the recurrence equation: $WC(v_i) = d(v_i) \times WC(v_j)$, if $d(v_i) > u$ and $WC(v_i) =$ *Constant* if $d(v_i) = u$. Hence the search time complexity is $O(n!/u!)$.

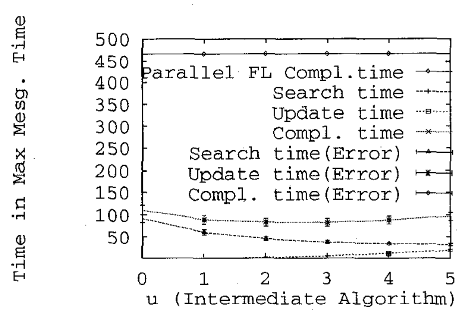
DEFINITION 11 (μ). *"Max. Message Time" or μ is the maximum time to send a message between two neighboring nodes.*

WC for Phases 2 and 3, for k nodes, is when the update messages for two nodes do not overlap. For messages to reach v_i, where $d(v_i) = u$, $WC(v_i) = O(k.u.\mu)$. However, due to concurrent updating, the average delays are far lower than the worst case costs.

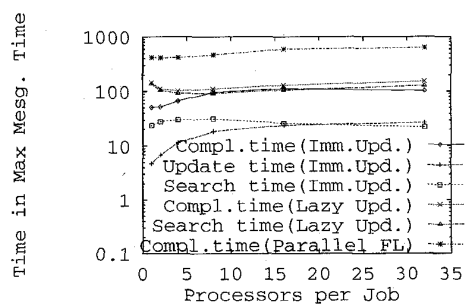
4.2 Simulations

We allocate 1,000 jobs on a 32 node hypercube. We assume the job arrival rate and service times to be Poisson and exponentially distributed, reflecting the memoryless property of the jobs, and subcube sizes to be uniformly distributed, with the service time of each *PN*, in the allocated subcube, same as the job service time. The jobs arrive at H_n and are queued up if they cannot be allocated *PN*s immediately. In *Immediate Update* search is conducted only when $avail(H_n) \ge k$. For other algorithms, on failure, search is repeated after some time. The performance measures are: t_a, the average completion time, t_s, the search time, t_u, the update time, p_u, the utilization of *PN*s, and queuing times (all plotted with 90% confidence intervals averaged over 20 runs, in terms of μ, except utilization). Let λ be the average interarrival time, η, the average completion time, and s_c, the mean subcube size of jobs.

Fig. 3a shows t_s, t_u, t_a as a function of u, in *Intermediate Update*. The end points correspond to the *Lazy* and *Immediate Update*. The assumptions are: $\lambda = 10\mu$, $\eta = 100\mu$, and $s_c = 8$ (three-dimensional). The startup transients are negligible with these parameters. When u increases, t_u increases but t_s decreases. Consequently, the t_a and queuing times shows a minimum between 0 and 5. As the updating costs dominate the search costs, p_u decreases with increase in u [9]. The search and update overheads are very low for λ, $\eta \sim \mu$.



(a)

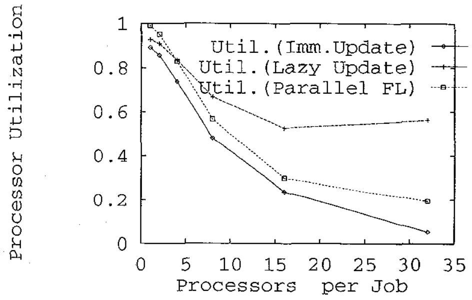


(b)

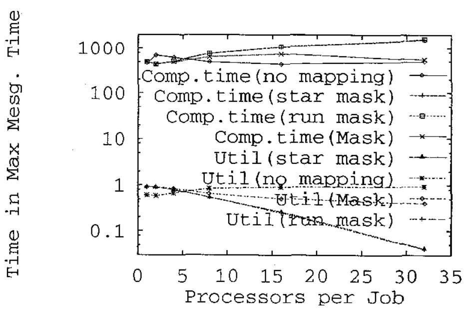
Fig. 3. Performance of (a) completion time with u, (b) completion time with s_c.

Fig. 3b shows t_s, t_u, t_a as a function of s_c. Jobs with large s_c have fine grain parallelism, and those with smaller s_c have coarse grain parallelism. The assumptions are: Total service time of a job is constant, i.e., $\eta_t \sim 1 / s_c$, where η_t is the mean service time of a task, $\lambda = 10\mu$, $\eta = 100\mu$ (for a job requiring all 32 nodes). Note that we omit confidence intervals to aid clarity in these graphs, and they are similar to Fig. 3a.

For *Lazy Update*, $t_u = 0$. With increase in s_c, t_s (and hence t_a) decreases (due to longer η_t and queued jobs), reaches a minimum, and then increases (due to inadequate available processors). For *Immediate Update*, t_u increases, and t_s increases and then decreases, with s_c. Hence t_a increases and then decreases. Here t_u increases due to larger *MN*s to be updated. WC is higher than simulation values, for t_s as compared to t_u (Fig. 5b). For *Immediate Update*, p_u decreases rapidly with s_c, as compared to *Lazy Update*, as more update messages block allocation of queued jobs (Fig. 4a). Hence *Lazy Update* is better for jobs with larger s_c, and *Immediate Update* for jobs with smaller s_c.



(a)



(b)

Fig. 4. Performance of (a) utilization with s_c, (b) completion time/utilization.

Adaptive Update adapts to *U* when *U* is difficult to estimate and may vary with time. t_u is lower when $\alpha_{v_i}^{low}$ is close to $\alpha_{v_i}^{high}$, as $\alpha_{v_i}^{curr}$ adapts faster to *U* [9]. However, a tightly constrained α has instability problems.

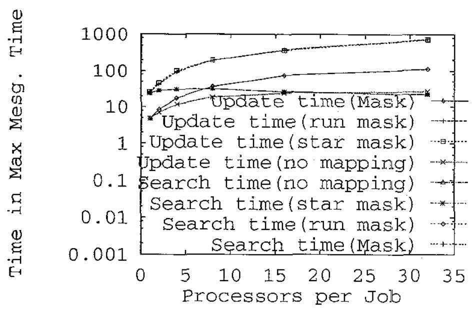
4.3 Performance of Update Algorithms under Various Mapping Strategies

For large λ and η (over 100,000 μ and 1,000μ), mapping overheads are negligible. Fig. 5a shows t_a with s_c for a sequential job (η = 4,000μ, λ = 1,000μ). Star mask and run mask have similar overheads. Mask mapping shows the best t_a, for small s_c. Mask mapping scheme and jobs with small s_c are suitable for mapped implementation.

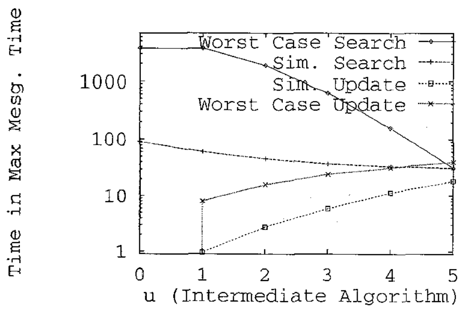
Update algorithms also perform better than sequential algorithms [9].

5 Conclusion

We have proposed an augmented architecture and algorithms, for efficient processor allocation and load balancing in binary hypercubes. The cost, performance and sensitivity of the algorithms indicate the following: *Intermediate Update* algorithm performs better than *Lazy* and *Immediate Update* algorithms, with *Lazy Update* suitable for fine grain parallelism and *Immediate Update* for coarse grain parallelism. *Adaptive Update* is useful when the job characteristics are not known or change with time.



(a)



(b)

Fig. 5. (a) Search/update time, (b) simulation vs. worst case.

Acknowledgments

We would like to thank Dr. Shahram Latifi and the referees for their useful comments.

References

[1] I. Ahmad, A. Ghafoor, and G.C. Fox, "Hierarchical Scheduling of Dynamic Parallel Computation on Hypercube Multicomputers," *J. Parallel and Distributed Computing*, vol. 20, no. 3, pp. 317–329, Mar. 1994.

[2] M.S. Chen and K.G. Shin, "Processor Allocation in a N-Cube Multiprocessor Using Gray Codes," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,396–1,407, Dec. 1987.

[3] P.J. Chuang and N.F. Tzeng, "A Fast Recognition-Complete Processor Allocation Strategy for Hypercube Computers," *IEEE Trans. Computers*, vol. 41, no. 4, pp. 467–479, Apr. 1992.

[4] W.J. Dally, "Virtual Channel Flow Control," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, Mar. 1992.

[5] S. Dutt and J.P. Hayes, "Subcube Allocation in Hypercube Computers," *IEEE Trans. Computers*, vol. 40, no. 3, pp. 341–352, Mar. 1991.

[6] P. Gaughan and S. Yalamanchili, "Adaptive Routing for Hypercube Interconnection Networks," *Computer*, pp. 12–23, May 1993.

[7] J. Kim, C.R. Das, and W. Lin, "A Processor Allocation Scheme for Hypercube Computers," *Proc. 1989 Int'l Conf. Parallel Processing*, vol. 2, pp. 231–238, Aug. 1989.

[8] D.W. Krumme, K.N. Venkataraman, and G. Cybenko, "Hypercube Embedding is NP-Complete," *Proc. Hypercube Multiprocessors 1986, PA: SIAM 1986*, pp. 148–157, 1986.

[9] H.K.B. Lalgudi, I.F. Akyildiz, and S. Yalamanchili, "Augmented Binary Hypercube: A New Architecture for Processor Management in Binary Hypercubes," TR-GIT/CSRL-93/03, pp. 27–34, Mar. 1993.

[10] D.D. Sharma and D.K. Pradhan, "Fast and Efficient Strategies for Cubic and Noncubic Allocation in Hypercube Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, pp. 118–127, 1993.

[11] Q. Yang and H. Wang, "New Graph Approach to Minimizing Processor Fragmentation in Hypercube Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 10, pp. 1,165–1,171, Oct. 1993.