# T U M

I N S T I T U T F Ü R I N F O R M A T I K


# Divide-and-Conquer Algorithms on the Hypercube


Ernst W. Mayr Ralph Werchner



TUM-I9322
August 1993



T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-8-1993-I9322-350/1.-FI


Typescript: ---

Druck: Mathematisches Institut und
Institut für Informatik der
Technischen Universität München

# Divide-and-Conquer Algorithms on the Hypercube


Ernst W. Mayr*
Institut für Informatik
Technical University
München, Germany

Ralph Werchner†
Fachbereich Informatik
J.W. Goethe-University
Frankfurt am Main, Germany


October 24, 1994


**Abstract**

We show how to implement divide-and-conquer algorithms without undue overhead on a wide class of networks. We give an optimal generic divide-and-conquer implementation on hypercubes for the class of divide-and-conquer algorithms for which the total size of the subproblems on any level of the recursion does not exceed the parent problem size. For this implementation, appropriately sized subcubes have to be allocated to the subproblems generated by the divide-steps. We take care that these allocation steps do not cause any unbalanced distribution of work, and that, asymptotically, they do not increase the running time. Variants of our generic algorithm also work for the butterfly network and, by a general simulation, for the class of hypercubic networks, including the shuffle-exchange and the cube-connected-cycles network. Our results can also be applied to optimally solve various types of routing problems.


**Topics:** *Theory of Parallel and Distributed Computation, Algorithms and Data Structures*


*`mayr@informatik.tu-muenchen.de`
†`werchner@informatik.uni-frankfurt.de`

# 1 Introduction

The divide-and-conquer approach has turned out to be a very successful programming paradigm. Especially in the field of parallel processing one of the most natural methods to solve problems is to divide them into subproblems, solve these in parallel and then recombine the solutions to a solution of the given problem instance.

The complete execution of a divide-and-conquer algorithm can be visualized as a tree, with the root representing the given instance of the problem and the other nodes representing subproblems. In this tree, we can associate with each node (subproblem) the set of processors dealing with the corresponding subproblem. From this point of view, the problem of efficiently allocating sets of processors to the subproblems (nodes of the tree) becomes evident. Our goal in this paper is to solve this problem in a way causing minimal overhead in addition to the running time needed by the divide- and conquer-steps themselves. Unlike for PRAM's [FW78] which provide complete processor interconnection and hence simple processor allocation mechanisms (but see also [vG89]), this problem is much more difficult on network architectures. Two major hurdles are (`i`) allocating appropriately sized subnetworks to subproblems and (`ii`) routing the subproblems to their subnetworks.

These two objectives are conflicting with each other. If we want to find appropriately sized subnetworks leaving only very few processors idle, the subproblems might have to be routed over long distances. On the other hand, keeping routes short may imply that not enough suitably sized subnetworks are available. We show that a compromise between these two extremes of the trade-off yields asymptotically optimal results for a wide class of divide-and-conquer algorithms.

Results concerning closely related allocation problems on the hypercube can be found in [CL90], [CS87], and [KDL89]. These papers consider the problem of allocating appropriately sized subcubes to incoming tasks in an on-line fashion on a hypercube that is already partially allocated. Our approach differs since we allocate subcubes to all subproblems on the same level of recursion simultaneously.

A number of papers consider implementations of divide-and-conquer where each divide or conquer step is executed by a single processor. Thus, if a divide step always creates two subproblems, the resulting dependency structure between processors is a binomial tree, whose embeddings into de Bruijn and other networks are discussed in [ZRL91]. In [SW92], the depth of this tree (which corresponds to the dimension of the required hypercube) is computed for a divide-and-conquer strategy yielding the minimal parallel running time.

# 2 Fundamental Concepts and Notation

## 2.1 Networks

Consider networks consisting of $p$ processors interconnected by bidirectional communication channels capable of transmitting $\Theta(\log p)$ bits per step. In particular, we consider the family of boolean hypercubes and the family of butterfly networks. Our main results also hold for the so called *hypercubic networks*, such as the shuffle-exchange network, the de-Bruijn network, and the cube-connected-cycles network, by a general simulation given in [S90].

A *d-dimensional hypercube* contains $2^d$ processors. Their id's are the strings in $\{0,1\}^d$. Two processors are connected iff their id's differ in exactly one bit. If we associate with each processor the integer represented by its id, the set of processors can be ordered according to this numbering. Let an interval $[a,b)$ of processors in a hypercube denote the set of processors with numbers $a, a+1, \ldots, b-1$. This interval is an $i$-dimensional subcube iff $2^i \,\Big|\, a$ and $a + 2^i = b$. In this paper we consider only subcubes which are intervals in this sense (though there are many more subcubes).

A *d-dimensional butterfly* contains $(d+1)\,2^d$ processors named by tupels from $[0,d] \times \{0,1\}^d$. Connections are between processors $(i,a)$ and $(i+1,b)$ iff $a = b$ or $a$ and $b$ differ only in position $i+1$. The set of processors $[0,i] \times [a,b)$ is an $i$-dimensional subbutterfly iff $2^i \,\Big|\, a$ and $a + 2^i = b$. By cutting all edges between the nodes of level $i$ and level $i+1$ the $d$-dimensional butterfly is disconnected into $2^{d-i}$ $i$-dimensional butterflies and $2^{i+1}$ $(d-i-1)$-dimensional butterflies. These subbutterflies will be used in our construction. We assume that the nodes in the butterfly network are ordered lexicographically, with the second component more significant than the first. An extensive collection of results concerning hypercubes and hypercubic networks can be found in [L92].

In the following sections we will focus on hypercube algorithms. In section 6 we then show how to modify our results for the butterfly network.

## 2.2 Restrictions on the Divide and Conquer-Implementations

We consider divide-and-conquer algorithms with the following properties:

- in the divide-step, each problem is split into subproblems with sizes adding up to no more than the size of the problem itself; size is measured in processor words;
- the solution to each subproblem is no longer than the subproblem itself;
- each subproblem is treated in a subcube at least its size. The input of the divide-step is assumed to be stored contiguously in a subcube (interval) starting at the first processor, and the output of a divide step is a sequence of subproblems each stored

in an interval of processors. The conquer-step works in an analogous manner. Note, however, that the intervals of subproblems are generally not subcubes.

Let the running time of the divide-step and the conquer-step for inputs of size $n$ be bounded by $T(n)$, let the sizes of the generated subproblems be bounded by $s(n)$, and let $c$ be the size of problems solved without any more recursive steps. Then an obvious lower bound for the running time of the divide-and-conquer algorithm is

$$\bar{T}(n) = T(n) + T(s(n)) + T(s(s(n))) + \ldots + T(c).$$

In the sequel we show how to reach this bound asymptotically for hypercubes and hypercubic networks.

From the properties we agreed on for the divide-step and the conquer-step, the following algorithmic problems become apparent:

- the generated subproblems in general don't match the sizes of subcubes (resp. subbutterflies);
- in general, subproblems are stored in intervals which are not subcubes; rather, the intervals have to be routed to subcubes.

We define the *subcube allocation problem with expansion $x$* as follows:

> Suppose we are given a sequence of intervals of total length $\leq n$ where each interval length is a power of 2. Then the intervals have to be aligned with subcubes in the interval $[0, xn)$. The algorithm has to be executed on a hypercube with a size of the next power of 2 greater than or equal $xn$.

# 3 The Subcube Allocation Problem

In this section we consider several subcube allocation algorithms for the hypercube. In the algorithms we use the following primitive operations for which logarithmic time algorithms are well-known:

- parallel-prefix (for a sequence $a_1, a_2, \ldots, a_p$ and an associative operator $\circ$, have, for all $i$, the $i$-th processor compute the value of $\bigcirc_{1\leq j\leq i}\, a_j$) [S80];
- pipelined parallel-prefix ($\log p$ independent parallel-prefix operations) [PM89];
- concentration routing, inverse concentration routing, monotone routing (each processor is the source and the destination of at most one data item, the order of the data items is preserved by the routing, and, in concentration routings, the data items are concentrated to the leftmost processors, one item per processor) [NS81].

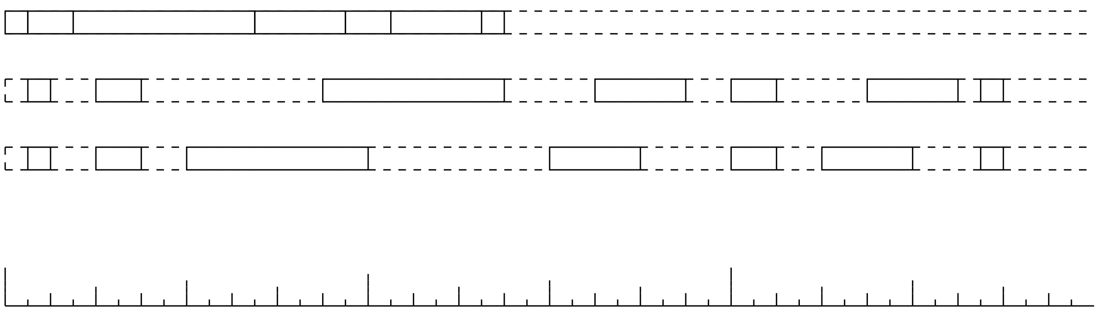


Figure 1: Subcube allocation by a monotone routing

**Lemma 1** *There is an $O(\log\log n \, \log n)$ time algorithm for the subcube allocation problem with expansion* 1*.*

**Proof:** It is sufficient to sort the given intervals according to their size in descending order, and to concentrate the resulting sequence to the left. As the length of each interval is a power of 2, a simple induction shows that the sorted intervals are aligned with subcubes.

Using parallel-prefix computations and monotone routings each data item computes the length of its associated interval. There are at most $\log n$ distinct lengths. The intervals are divided into groups according to their lengths. Using parallel-prefix operations and concentration routings, the upper half of groups is concentrated to the left and the lower half to their right. This process is repeated recursively within the two halves until the intervals are sorted according to their lengths. The depth of the recursion is bounded by $\log\log n$. ∎

If the running time of the given divide-step and conquer-step is only logarithmic in $n$ the above method leads to a suboptimal algorithm due to the $\log\log n$ factor. Our next lemma shows how to achieve an optimal running time at the cost of a somewhat larger expansion.

**Lemma 2** *There is an $O(\log n)$ time algorithm for the subcube allocation problem with expansion* 2*.*

**Proof:** Assume that the sequence of intervals (each with length a power of 2) with a total length of $n$ is stored in the leftmost processors of a hypercube of size at least $2n$. We first route the intervals to the right so that immediately to the left of each interval there is empty space equal to the size of the interval. Then each interval is shifted to the left until it is aligned with an appropriately sized subcube. In this shift operation no two intervals interfere with each other (see Figure 1).

It is obvious that the final destinations of the intervals can be computed in advance and only one inverse concentration route has to be carried out. ∎

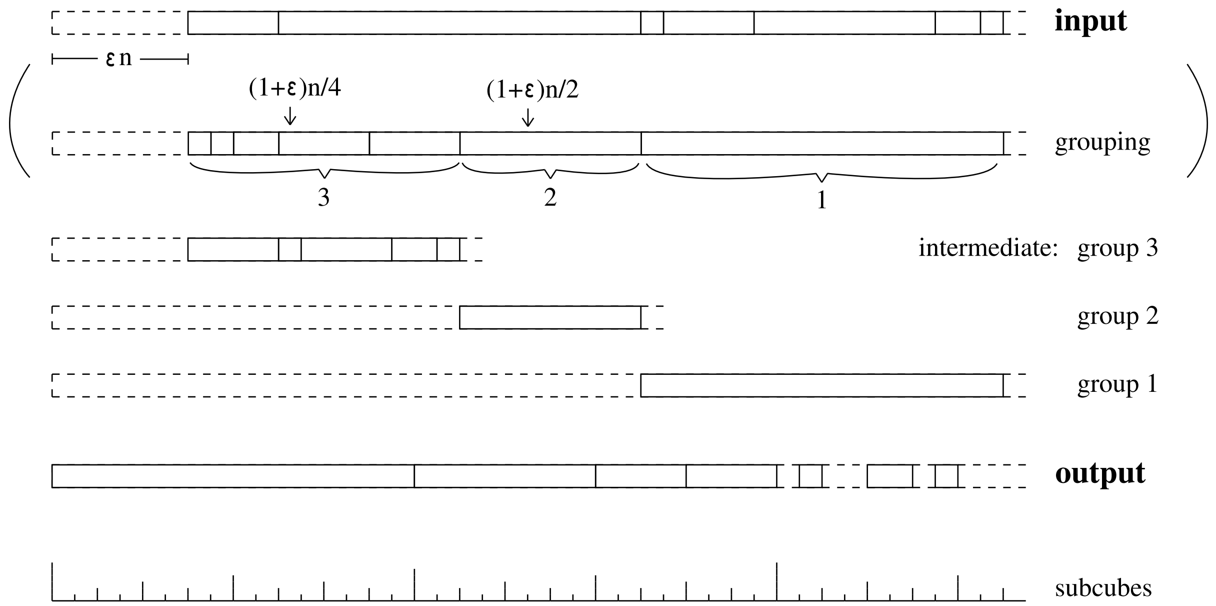


Figure 2: Subcube allocation in groups

The following result shows that the two previous lemmata can be viewed as the extreme points of a trade-off between the expansion and the time required for the routing. For notational convenience let $\log x$ be 1 for $x \leq 2$.

**Theorem 3** *The subcube allocation problem with expansion $1+\varepsilon$ can be solved in $O(\log\log 1/\varepsilon \ \log n)$ steps, for $\frac{1}{n} \leq \varepsilon \leq 1$.*

**Proof:** The main observation for this algorithm is that in an embedding of large intervals by the previous method the gaps between the intervals can be used to embed smaller intervals. Furthermore, this process can be repeated.

Let $g_i$ be the total size of all intervals of length $2^i$ for $0 \leq i \leq \lfloor \log n \rfloor$, and let

$$\begin{aligned} G_i &= \varepsilon\, n + \sum_{0 \leq k \leq i} g_k \\ r &= \lceil \log(1/\varepsilon + 1) \rceil \\ h_j &= \min\left\{ i \,\middle|\, G_i \geq (1+\varepsilon)n/2^j \right\} \qquad (0 \leq j < r) \end{aligned}$$

We divide the intervals into $r$ groups (see Figure 2). The $j$-th group contains all intervals of size $a$ with $2^{h_j} < a \leq 2^{h_{j-1}}$ (setting $h_r = -\infty$). The groups can be identified, separated and sorted in $O(\log r \log n)$ steps by the method given in the proof of Lemma 1.

Assume that we have a hypercube of size at least $\sum_{1 \leq j \leq r} G_{h_j}$ processors (we can simulate such a cube by the given cube with constant slowdown since $\sum_{1 \leq j \leq r} G_{h_j} < 3(1+\varepsilon)n$). We insert empty space between the groups so that for the $j$-th group an interval of length $G_{h_j}$

is available. Using the method given in the proof of Lemma 2 (treating the largest intervals separately) an embedding for each group in $[0, G_{h_j})$ is computed.

Now we use the observation mentioned at the beginning of this proof. Suppose we have two embeddings of the following form:

The first embedding maps a set of intervals, each of size at least $2^i$, to an interval $[0, a)$ leaving empty space $b$; the second embedding maps intervals of size less than $2^i$ in the interval $[0, b)$ leaving empty space $c$.

Then we can merge, in a straightforward way and using two monotone routings, the two embeddings into one embedding of all intervals in the interval $[0, a)$ leaving empty space $c$.

Applying this step repeatedly, the $r$ embeddings computed above are merged into one embedding of all intervals in the interval $[0, (1+\varepsilon)n)$. This process requires $O(\log r \log n)$ steps. ■

It turns out that it is also possible to embed intervals into subcubes with expansion 1 using only logarithmic time. For this algorithm we use our generic divide-and-conquer method presented in the following section as a bootstrap.

# 4 A Generic Divide-and-Conquer Algorithm

## 4.1 A Suboptimal Solution

As already noted in the introduction a divide-and-conquer algorithm can be viewed as a tree with the root corresponding to the given problem, the other nodes corresponding to the subproblems, and the leaves corresponding to subproblems solved directly. Our first implementation of the divide-and-conquer paradigm proceeds in phases corresponding to the levels of the tree. Each phase consists of the following steps:

A1. All subproblems at the current level of the tree are padded in size to the next larger power of 2 and are stored contiguously from the left of the hypercube. Using the algorithm of Lemma 2 each subproblem is routed to its own subcube;

A2. for each subproblem the divide-step is executed within its associated subcube.

When all subproblems are of some constant size, they are solved directly. In the ascending ("conquer") part of the algorithm we recombine these solutions to the subproblems. To guarantee correct combination of the computed results we have to remember the data movements made in the subcube allocation routines of the descending part, and reverse them. The data movements in the descending part of the algorithm consist of a single monotone routing per phase. If each processor stores whether or not it was the source of a data item, this routing can be reversed in logarithmic time. Thus, we only need a constant number of bits per processor and phase to store the moves. The reverse routing can be simplified at the cost of some additional memory requirements by storing, for each time step, which neighboring processor sent a data item.

Note that for the algorithm to work correctly the size of the hypercube must be at least $4n$ for inputs of size $n$.

Each of the subcube allocation steps in the above algorithm takes $O(\log n)$ steps. Thus, the total running time is

$$T(n) + O(\log n) + T(s(n)) + O(\log n) + T(s(s(n))) + \ldots + O(\log n) + T(c)$$

where again $T(n)$ is a bound on the running time of the divide-step and the conquer-step, $s(n)$ is the maximum size of a subproblem, and $c$ is the size of problems solved directly.

Since the time required for the subcube allocation steps does not decrease on deeper levels of the recursion, this running time may be significantly larger than the lower bound $\bar{T}(n)$. In our next algorithm we manage to resolve this deficiency by restricting the subproblem allocation routings to smaller subcubes.

To solve this problem, one could try to allocate subcubes to subproblems generated from a problem $a$ only within the subcube allocated to $a$. But in this way the maximum load of subcubes on each level of the recursion would, in the worst case, grow exponentially with the depth. A better solution is to allocate subcubes within ranges that are small enough to use fast allocation algorithms, and large enough to bound the growth of unbalance in the distribution of the data items on deeper levels of the recursion. The details of this method are presented in the following subsection.

## 4.2 An Optimal Solution

We describe the descending part of our optimal algorithm. For an $i$-dimensional cube $C$, which is initially the whole cube, the algorithm consists of the following steps:

1. Execute steps A1 and A2 on $C$ until all subproblems are smaller than $2^{\lfloor i/4 \rfloor}$.
2. Divide the subproblems in $C$ into $2^{\lceil i/2 \rceil}$ roughly equally sized groups as follows: Concentrate all intervals comprising the subproblems to the first processors in $C$, say $[a, a+n')$. For all $j$ with $0 \leq j < 2^{\lceil i/2 \rceil}$, assign to group $j$ all intervals beginning in

$$\left[ a + \left\lfloor j \frac{n'}{2^{\lceil i/2 \rceil}} \right\rfloor \; , \, a + \left\lfloor (j+1) \frac{n'}{2^{\lceil i/2 \rceil}} \right\rfloor \right) .$$

   Route the intervals of group $j$ into the $j$-th $\lfloor i/2 \rfloor$-dimensional subcube of $C$, i.e. the subcube starting at processor $a + j\, 2^{\lfloor i/2 \rfloor}$. Call this algorithm recursively for each of these subcubes provided that their dimension $\lfloor i/2 \rfloor$ is not below some appropriately chosen constant $d_0$. Otherwise, all subproblems are of constant size and are solved directly.

To guarantee that the subcube allocation steps in the phases can be executed correctly, we had to assume in the previous subsection that the cube was at least 4 times as large as the input size, or, as we shall say subsequently, the initial load of the cube was at most 1/4.

**Lemma 4** *With an initial load* $\leq 0.183$ *and* $d_0 = 8$*, the above algorithm is never invoked on a subcube with a load* $> 1/4$.

**Proof:** Consider an $i$-dimensional subcube $C$ invoking, in step 2, the algorithm recursively for an $i'$-dimensional subcube $C'$. Let $l(C)$ and $l(C')$ denote the load of $C$ and $C'$, then we have

$$\begin{aligned} l(C') &< \left( \frac{l(C)\,2^i}{2^{i-i'}} + 2^{\lfloor i/4 \rfloor} \right) \frac{1}{2^{i'}} \\ &= l(C) + 2^{\lfloor i/4 \rfloor - i'} \\ &= l(C) + 2^{\lfloor i/4 \rfloor - \lfloor i/2 \rfloor} \\ &\leq l(C) + 2^{-i'/2} \end{aligned}$$

Thus, for an $i$-dimensional subcube, for which the algorithm is invoked recursively starting from a $d$-dimensional hypercube, the load is bounded by

$$\begin{aligned} & 0.183 + 2^{-\lfloor d/2 \rfloor /2} + 2^{-\lfloor \lfloor d/2 \rfloor /2 \rfloor /2} + \ldots + 2^{-i/2} \\ \leq \; & 0.183 + \sum_{j \geq 0} \left( 2^{-i/2} \right)^{(2^j)} \end{aligned}$$

An easy computation shows that this sum is less than 1/4 for $i \geq 8$. ■

To analyze the running time of this divide-and-conquer algorithm, we only have to consider the descending part, because the ascending part is symmetrical.

Let us at first estimate the running time of step 2, the grouping and routing of subproblems. These operations need $O(i)$ steps in an $i$-dimensional subcube. But as the dimension of the subcubes is halved on each level of recursive calls, the sum of these running times is $O(\log n)$.

Next, consider step 1. The time needed for the subcube allocation algorithm (step A1) executed in an $i$-dimensional subcube is $O(i)$. We compare this time with the running time of the immediately following divide-step (step A2) in the largest allocated subcube, which must be at least of size $2^{\lfloor i/4 \rfloor}$ by our termination criterion for the loop of step 1. Provided that the running time of the divide-step or conquer-step is $T(n) = \Omega(\log n)$, we spend $\Omega(i)$ steps for this divide-step or the corresponding conquer-step. Therefore, the subcube allocation steps cause only a constant factor overhead in the running time.

We should note here that the authors could hardly think of a divide-step and conquer-step for any nontrivial problem running in $o(\log n)$ time on the hypercube.

**Theorem 5** *Given a divide-step and a conquer-step satisfying the assumptions listed in section 2, with running times $T(n) = \Omega(\log n)$ on the hypercube, there is an algorithm solving problems of size $n$ on a $\lceil \log n \rceil$-dimensional hypercube in time $O(\bar{T}(n))$, with*

$$\bar{T}(n) = T(n) + T(s(n)) + T(s(s(n))) + \ldots + T(c),$$

*for some constant $c$ and a bound $s(n)$ on the size of subproblems.*

**Proof:** We have given an algorithm achieving the time bound on a hypercube with an initial load of at most 0.183. This larger hypercube can be simulated by a $\lceil \log n \rceil$-dimensional hypercube with a slowdown of at most 8. ∎

# 5 Applications

In [MW92] a divide-and-conquer algorithm is given that performs a special class of partial permutations called *parentheses-structured routings* in logarithmic time on hypercubes. In a well-formed string of parentheses, these partial permutations map each opening parenthesis to its matching closing parenthesis. The method used in [MW92] employs specific properties of this problem in order to obtain an efficient implementation of the algorithm. The generic approach presented in this paper provides an alternative implementation with the same running time.

As another application of Theorem 5, we present a subcube allocation algorithm with expansion 1 for hypercubes which runs in logarithmic time.

Let $d$ be the dimension of the cube. We execute the following steps (see Figure 3):

1. Separate the *small* intervals of size $< 2^{\lceil d/3 \rceil}$ from the *large* intervals concentrating the large intervals to the left and the small intervals to their right.

2. Let $n'$ denote the total size of the large intervals, and set $n'' = n'/2^{\lceil d/3 \rceil}$. Redistribute the large intervals by performing an *unshuffle permutation*, i.e. route the data item stored in processor $i + j\,2^{\lceil d/3 \rceil}$ (for $0 \le i < 2^{\lceil d/3 \rceil}$ and $0 \le j < n''$) to processor $j + i\,n''$, using a bit-permute-permutation [NS82a] and a concentration routing.
   Now consider each of the intervals $[i\,n'', (i+1)\,n'')$ as a new subproblem *for this subcube allocation algorithm*, whose corresponding (and identical) subsolutions can be combined by reversing the above routing.

3. Cut the sequence of small intervals into groups of contiguous intervals, each of length between $2^{\lfloor \frac{2}{3} d \rfloor}$ and $2^{\lfloor \frac{2}{3} d \rfloor} + 2^{\lceil d/3 \rceil}$ (only the last group may be smaller). This can easily be done by cutting near multiples of $2^{\lfloor \frac{2}{3} d \rfloor} + 2^{\lceil d/3 \rceil - 1}$.

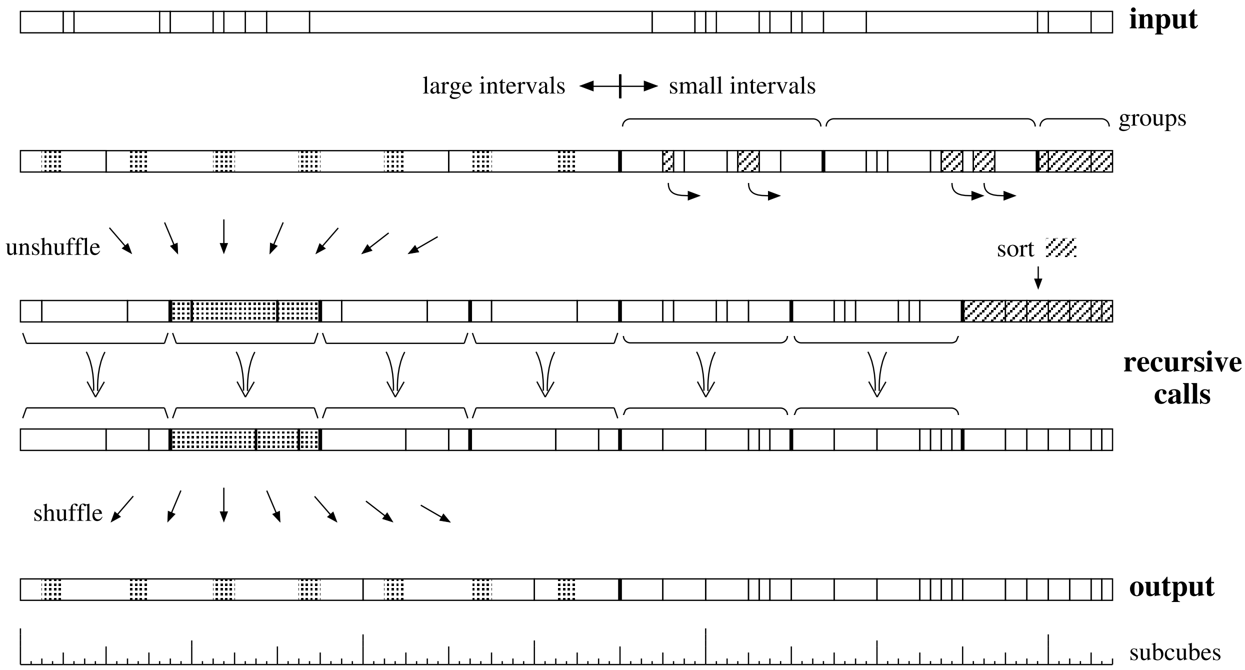


Figure 3: Recursive subcube allocation algorithm

4. Use a pipelined segmented parallel prefix operation to find the distribution of interval sizes within the groups. Then, in each group of size $s$ mark a set of intervals with a total size of $s - 2^{\lfloor \frac{2}{3}d \rfloor}$.

5. Concentrate the marked intervals to the right. Sort them together with the last group in order of descending size. Since there are $O(2^{\frac{2}{3}d})$ data items to be sorted, this step can be performed in logarithmic time using *sparse enumeration sort* [NS82b].

   Again note that each group of size $2^{\lfloor \frac{2}{3}d \rfloor}$ comprises a subproblem; after solving these subproblems, all small intervals are embedded into subcubes.

The divide-step and the conquer-step just outlined can both be implemented in logarithmic time on a $d$-dimensional hypercube. All generated subproblems are of size $\leq 2^{\lfloor \frac{2}{3}d \rfloor}$. Thus we can apply Theorem 5 with

$$T(n) = O(\log n), \quad s(n) \leq n^{2/3}, \quad \text{and hence} \quad \bar{T}(n) = O(\log n),$$

and obtain:

**Theorem 6** *There exists an $O(\log n)$ time algorithm for the subcube allocation problem with expansion* 1.

# 6 The Butterfly

For the butterfly network we require two modifications of the method presented in section 4.

The first modification is due to the fact that butterflies cannot easily be divided into subbutterflies. In step 2 of the optimal divide-and-conquer implementation, a butterfly network of dimension $2\,i+1$ is divided into $2^{i+2}$ $i$-dimensional butterfly networks whereas a butterfly of dimension $2\,i$ has to be divided into $2^{i}$ $i$-dimensional and $2^{i+1}$ $(i-1)$-dimensional butterflies. The grouping of subproblems has to be adjusted accordingly so as to distribute the load roughly equally to the subbutterflies.

Considering step 1, the subnetwork allocation problem appears to be slightly more complicated for the butterfly than for the hypercube. There is a solution to the subnetwork allocation problem for the butterfly with constant expansion. But the task of routing each subproblem into its allocated subbutterfly appears to be much easier if we additionally require that subproblems be at least of size $(\lfloor i/4 \rfloor + 1)\,2^{\lfloor i/4 \rfloor}$ in an $i$-dimensional butterfly.

For the following lemma assume that the intervals to be embedded have lengths matching the sizes of butterflies.

**Lemma 7** *The subnetwork allocation problem with expansion* 8 *for butterflies can be solved in* $O(\log n)$ *steps if each of the intervals given in a* $d$*-dimensional butterfly is at least of size* $(\lfloor d/4 \rfloor + 1)\,2^{\lfloor d/4 \rfloor}$.

**Proof:** Let a sequence of intervals be given, and let the length of the $j$-th interval be $(d_j+1)\,2^{d_j}$. Since the expansion is at least 8, we have

$$\begin{array}{rrcl} & \sum_j (d_j+1)\,2^{d_j} & \leq & \frac{1}{8}(d+1)\,2^d \\ \Longrightarrow & \sum_j 2^{d_j} & \leq & \frac{1}{2}\,2^d \end{array}$$

Using the simple technique of Lemma 1, we can embed a sequence of intervals with lengths $2^{d_j}$ into subcubes of a $d$-dimensional hypercube without changing the order of the intervals. If, instead, in the hypercube the $j$-th interval is embedded in $[a_j, a_j + 2^{d_j})$, then we solve our problem for the butterfly by routing the $j$-th interval to the set of processors $[0, d_j] \times [a_j, a_j + 2^{d_j})$. This routing is monotone in the chosen ordering of the $d$-dimensional butterfly and can be performed in logarithmic time. ■

For the $i$-dimensional butterfly we substitute the upper bound for the subproblem sizes that we could guarantee after step 1 by $(\lfloor i/4 \rfloor + 1)\,2^{\lfloor i/4 \rfloor}$. We can also achieve this bound by collecting, after each execution of steps A1 and A2, those subproblems that are already smaller than $(\lfloor i/4 \rfloor + 1)\,2^{\lfloor i/4 \rfloor}$ to one end of the butterfly and subdividing further only those subproblems that are still too large. In this way we can guarantee the condition of Lemma 7.

A proof very similar to that of Theorem 5 (and therefore omitted) yields:

**Theorem 8** *We are given a divide-step and a conquer-step for a problem $P$ satisfying the assumptions given in section 2 with running times $T(n) = \Omega(\log n)$ on a butterfly of size $b(n)$, where $b(n)$ is the smallest size $\geq n$ of a butterfly. Then $P$ can be solved for instances of size $n$ on a butterfly of size $b(n)$ in time $O(\bar{T}(n))$.*

There are many networks with a structure very similar to the butterfly. These *hypercubic networks* include the cube-connected-cycles network, the Benes network, the shuffle-exchange network, and the de Bruijn network. The analogue of Theorem 8 is valid for each of these networks. This follows from a result shown by Schwabe in [S90] stating that a network of one of these types can be simulated by a similar size network of any other type with a constant factor slowdown. To obtain Theorem 8 for some family $F$, we have to simulate the divide-step and the conquer-step by a butterfly, apply Theorem 8, and simulate the complete butterfly divide-and-conquer implementation on the given network of type $F$.

A direct proof of Theorem 8 for each of the hypercubic networks seems to be difficult because the subnetwork allocation problem can become quite complicated. Indeed, for the cube-connected-cycles network, it is unsolvable since no connected regular network can contain a regular subnetwork of the same degree.

# 7 Conclusion and Open Problems

In this paper we have implemented divide-and-conquer on the hypercube and the hypercubic networks with an asymptotically optimal running time. If the goal is minimizing the constant factor in the overhead caused by our method, there are a number of design decisions that can be made, depending on the parameters $T(n)$ and $s(n)$. We have presented different subcube allocation algorithms, and one could use different subcube sizes for the recursive calls and different parameters for allocating subcubes. The choice of parameters given in this paper works relatively well in all cases. Indeed, it is not even necessary to know the bounds $T(n)$ and $s(n)$.

If $T(n) = \omega(\log\log n \log n)$ the simple subcube allocation algorithm with expansion 1 of Lemma 1 should be used. Compared with the implementation given in section 4 this choice improves the processor utilization by a factor of 2 while increasing the time for subcube allocation to an amount still negligible compared to the total running time. In this way, the slowdown in the final simulation of the sufficiently large cube by the given cube is halved and the total running time is nearly halved. Further improvements in processor utilization can be achieved if, for the allocation of subcubes, the whole cube is considered on each level of recursion, which was our first "suboptimal" solution. Again, this implementation should be chosen if $\bar{T}(n)$ is sufficiently large.

A more general (and, of course, more complicated) approach to the problems considered in this paper would be to allow the total size of subproblems generated on any one level to exceed the original problem size and/or to allow the number of processors performing a divide-step or conquer-step to depend non-linearly on the problem size.

Another interesting open question is to find a direct optimal algorithm for the subcube allocation problem with expansion 1.

# References


[CL90] M.Y. Chan and S.-J. Lee. Subcube recognition, allocation/deallocation and relocation in hypercubes. *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing,* 87–93, 1990.

[CS87] M.-S. Chen and K.G. Shin. Processor allocation in an $N$-cube multiprocessor using Gray codes. *IEEE Transactions on Computers,* C-36:1396–1407, 1987.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. *Proceedings of the 10th ACM Symposium on Theory of Computing,* 114–118, 1978.

[vG89] A. van Gelder. PRAM processor allocation: A hidden bottleneck in sublogarithmic algorithms *IEEE Transactions on Computers,* C-38:289–292, 1989.

[KDL89] J. Kim and C.R. Das and W. Lin. A processor allocation scheme for hypercube computers. *Proceedings of the 1989 International Conference on Parallel Processing.* Vol. 2 Software, 231–238, 1989.

[L92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers, 1992.

[MW92] E.W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures,* 109-117, 1992.

[NS81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers,* C-30:101–107, 1981.

[NS82a] D. Nassimi and S. Sahni. A self-routing Benes network and parallel permutation algorithms. *IEEE Transactions on Computers,* C-31:148–154, 1982.

[NS82b] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *JACM,* 29:642–667, 1982.

[PM89] C.G. Plaxton and E.W. Mayr. Pipelined parallel prefix computations and sorting on a pipelined hypercube. *J. Parallel Distrib. Comput..* 17:374–380, 1993.

[S90] E.J. Schwabe. On the computational equivalence of hypercube-derived networks. *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures,* 388-397, 1990.

[S80] J.T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems,* 2:484–521, 1980.

[SW92] H. Sethu and M.D. Wagh. Design of time-optimal hardware-efficient divide-and-conquer algorithms. *Proceedings of the 1992 International Conference on Parallel Processing.* Vol. 3 Algorithms & Applications, 237–240, 1992.

[ZRL91] X. Zhong, S. Rajopadhye and V.M. Lo. Parallel implementations of divide-and-conquer algorithms on binary de Bruijn networks. Technical Report CIS-TR-91-21 University of Oregon, 1991.