Concatenated Parallelism: A Technique for Efficient Parallel Divide and Conquer[1]

(*Preliminary Version*)

Srinivas Aluru, Sanjay Goil[2]
School of CIS and
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244-4100
email: *aluru, sgoil@top.cis.syr.edu*

Sanjay Ranka[3]
School of CISE

University of Florida
Gainesville, FL 32611
ranka@cis.ufl.edu

[1]We are grateful to Northeast Parallel Architectures Center and Minnesota Supercomputing Center for allowing us to use their CM-5. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[2]Supported in part by NASA under subcontract #1057L0013-94 issued by the LANL.

[3]Supported in part by NSF under ASC-9213821, AFMC and ARPA under contract #F19628-94-C-0057 and the University of Florida.

Abstract

A number of problems have efficient algorithms that are based on the divide and conquer paradigm. Such problems can be solved in parallel by mapping the corresponding divide and conquer tree to the parallel computer under consideration. Two basic strategies are used in such parallelizations: *Task Parallelism*, in which different subproblems are assigned to different groups of processors and *Data Parallelism*, in which the tasks are solved one after the other using all the processors. Task parallelism involves significant data movement and data parallelism causes problems due to load imbalance.

In this paper we propose a new strategy, which we call *Concatenated Parallelism*, for efficient parallel solution of problems resulting in divide and conquer trees. Our strategy is useful when the communication time due to data movement in distributing the subproblems using task parallelism is significant when compared to the time required to divide the subproblems. This happens to be the case for a number of important applications including quicksort, quickhull and the construction of quadtrees, octrees and multidimensional binary search trees. We consider both balanced and unbalanced deterministic divide and conquer trees as well as randomized divide and conquer trees. We provide both theoretical and experimental analysis of Concatenated Parallelism for coarse-grained distributed memory parallel machines along with comparisons with task and Data Parallelism. We have implemented our algorithms on the CM-5 and report on the experimental results. One useful application of our technique is a scalable parallel quicksort algorithm.

1 Introduction

Scheduling a number of tasks on a parallel machine to minimize the running time for the completion of all the tasks is a well-studied problem in parallel computing. In the most general case, the tasks can be of varying sizes and each task itself can be solved in parallel. Two basic types of parallelism can be exploited: Scheduling of independent tasks to different groups of processors such that the tasks can be solved simultaneously in parallel is called *Task Parallelism*. Solving each individual task in parallel using all the processors and solving the tasks one after the other is called *Data Parallelism*. Of course, it is possible to use a combination of these strategies for optimal scheduling, and such a strategy is referred to as *Mixed Parallelism*. Several researchers have worked on exploiting mixed parallelism, both in theory [3, 6, 11, 17] and in practice [4, 5, 13, 16].

In a number of problems, all the tasks may not be known in advance but may be generated dynamically as existing tasks are processed. This is the case with problems whose efficient solutions use the divide and conquer strategy. The execution of an instance of such a problem can be represented by a divide and conquer tree. Each internal node of the tree corresponds to a task. After performing some computation, the task is split (divide step) into several subtasks which are represented by the children of the node. The subtasks are solved recursively and the solutions may need to be combined to find the solution for the task (merge step).

Several important issues arise in parallelizing such applications using task and data parallelism. Suppose that a task is currently distributed on a group of processors. After performing the work required to divide the task into subtasks in parallel, several options exist for the solution of the subtasks. In the task parallelism approach, the processors are divided into subgroups, perhaps according to the size of the subtasks, and the subtasks are moved to their respective subgroups of processors and solved independently. This requires movement of data to the appropriate processor subgroup. In the data parallelism approach, the subtasks are solved one after another using all the processors. Unfortunately, each subtask may not be uniformly spread across the processors even though the parent task is. Hence, data parallelism may lead to severe load imbalance. Also, the practical efficiency of a parallel algorithm often decreases with increase in the number of processors. If the time required to divide the subtasks is significantly higher than the cost of redistribution, communication time due to allocation of the subtasks can be ignored. Such an assumption is often made in the literature [4]. Unfortunately, it is not valid for several important problems, which include quicksort, quickhull, construction of quad/octrees and multidimensional binary search trees.

In this paper, we propose a new strategy called *Concatenated Parallelism* for efficient solution of problems resulting in divide and conquer trees. The basis idea is to solve all the subtasks together using all the processors. Even though the distribution of each subtask is non-uniform, this scheme does not lead to load imbalance (as in data parallelism) because the sum of the sizes of the subtasks allocated to each processor is uniform. This scheme also eliminates the communication due to data movement in the intermediate steps, the drawback of task parallelism. The strategy is particularly useful when the sizes of the subtasks may be non-uniform. The only disadvantage of concatenated parallelism is that communication in solving the subtasks involves all the processors as opposed to increasingly smaller subsets as in task parallelism. However, we can often considerably reduce this expense by spooling the communication required for all the subtasks. Such a strategy significantly reduces the communication cost because set up times for communication are typically higher than transmission costs by two orders of magnitude. We use the concatenated parallelism strategy until enough subtasks are generated to map them uniformly to individual processors. At this stage, one redistribution is performed followed by sequentially solving the subtasks.

Our focus in this paper is in designing strategies that have practical efficiency on parallel computers. Therefore, we use coarse-grained distributed memory parallel computers as our models of parallel computation as most existing parallel computers belong to this category. A coarse-grained parallel computer consists of several relatively powerful processors connected by an interconnection network. Instead of making specific assumptions about the network connecting processors, we describe our algorithms in terms of some basic communication primitives. The running time of our algorithms on a specific interconnection network can be easily derived by substituting the running times of the communication primitives. We provide such an analysis for hypercubes and meshes.

The rest of the paper is organized as follows: In Section 2, we describe our model of parallel computation and outline some primitives used by our algorithms. In Section 3, we describe concatenated parallelism. In Sections 4, 5 and 6, we apply Concatenated Parallelism to three types of divide and conquer trees: deterministic trees resulting in balanced subtasks, deterministic trees resulting in non-uniform sized subtasks and randomized trees. For each type, we present sample applications along with experimental results on the CM-5. The performance of concatenated parallelism is compared with both task and data parallelism. Section 7 contains various redistribution strategies that can be used for distributing subtasks to individual processors. We conclude the paper in Section 8.

2 Model of Parallel Computation

Coarse Grained Machines (CGMs) consist of a set of processors (tens to a few thousand) connected through an interconnection network. The memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space. Popular interconnection topologies are buses (SGI Challenge), 2D meshes (Paragon, Delta), 3D meshes (Cray T3D), hypercubes (nCUBE), fat tree (CM5) and hierarchical networks (cedar, DASH).

CGMs have cut-through routed networks which will be used for modeling the communication cost of our algorithms. For a lightly loaded network, a message of size m traversing d hops of a cut-through (CT) routed network incurs a communication delay given by $T_{comm} = t_s + t_h d + t_w m$, where t_s represents the handshaking costs, t_h represents the signal propagation and switching delays and t_w represents the inverse bandwidth of the communication network. The startup time t_s is often large, and can be several hundred machine cycles or more. The per-word transfer time t_w is determined by the link bandwidth. t_w is often higher (an order to two orders of magnitude is typical) than t_c, the time to do a unit computation on data available in the cache. The per-hop component $t_h d$ can often be subsumed into the startup time t_s without significant loss of accuracy. This is because the diameter of the network, which is the maximum of the distance between any pair of processors, is relatively small for most practical sized machines, and t_h also tends to be small. The above expressions adequately model communication time for lightly loaded networks. However, as the network becomes more congested, the finite network capacity becomes a bottleneck. Multiple messages attempting to traverse a particular link on the network are serialized. A good measure of the capacity of the network is its cross-section bandwidth (also referred to as the bisection width). For p processors, the bisection width is $\frac{p}{2}$, $2\sqrt{p}$, and 1 for a hypercube, wraparound mesh and for a shared bus respectively.

Our analysis will be done for the following interconnection networks: hypercubes and two dimensional meshes. The analysis for permutation networks and hypercubes is the same in most cases. These cover nearly all commercially available machines. A permutation network is one for which almost all of the permutations (each processor sending and receiving only one message of equal size) can be completed in nearly the same time (e.g. CM-5 and IBM SP Series).

Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures. Several algorithms have been described in the literature for these primitives and are part of standard textbooks [7, 9]. The use of collective communication provides a level of architecture independence in the algorithm design. It also allows for precise analysis of an algorithm by replacing the cost of the primitive for the targeted architecture.

In the following, we describe some important parallel primitives that are repeatedly used in our algorithms and implementations. For commonly used primitives, we simply state the operation involved. The analysis of the running time is omitted and the interested reader is referred to [9]. For other primitives, a more detailed explanation is provided. Table 1 describes the collective communication routines used in the development of our algorithms and their time requirements on cut-through routed hypercubes and meshes. In what follows, p refers to the number of processors.

1. **Broadcast.** In a Broadcast operation, one processor has a message of size m to be broadcast to all other processors.

2. **Combine.** Given a vector of size m on each processor and a binary associative operation,

Primitive	Running time on a p processor	
	Hypercube	Mesh
Broadcast	$O((t_s + t_w m)\log p)$	$O((t_s + t_w m)\log p + t_h\sqrt{p})$
Combine	$O((t_s + t_w m)\log p)$	$O((t_s + t_w m)\log p + t_h\sqrt{p})$
Parallel Prefix	$O((t_s + t_w)\log p)$	$O((t_s + t_w)\log p + t_h\sqrt{p})$
Gather	$O(t_s \log p + t_w mp)$	$O(t_s \log p + t_w mp + t_h\sqrt{p})$
Global Concatenate	$O(t_s \log p + t_w mp)$	$O(t_s \log p + t_w mp + t_h\sqrt{p})$
All-to-All Communication	$O((t_s + t_w m)p + t_h p \log p)$	$O((t_s + t_w mp)\sqrt{p})$
Transportation Primitive	$O(t_s p + t_w r + t_h p \log p)$	$O((t_s + t_w r)\sqrt{p})$
Order Maintaining Data Movement	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$
Non-order Maintaining Data Movement	$O(t_s p + t_w(s_{max} + r_{max}) + t_h p \log p)$	$O((t_s + t_w(s_{max} + r_{max}))\sqrt{p} + t_h\sqrt{p})$

Table 1: Running times of various parallel primitives on cut-through routed hypercubes and square meshes with p processors.

the Combine operation computes a resultant vector of size m and stores it on every processor. The i^{th} element of the resultant vector is the result of combining the i^{th} element of the vectors stored on all the processors using the binary associative operation.

3. **Parallel Prefix.** Suppose that $x_0, x_1, \ldots, x_{p-1}$ are p data elements with processor P_i containing x_i. Let $\otimes$ be a binary associative operation. The Parallel Prefix operation stores the value of $x_0 \otimes x_1 \otimes \ldots \otimes x_i$ on processor P_i.

4. **Gather.** Given a vector of size m on each processor, the Gather operation collects all the data and stores the resulting vector of size mp in one of the processors.

5. **Global Concatenate.** This is the same as Gather except that the collected data should be stored on all the processors.

6. **All-to-All Communication.** In this operation each processor sends a distinct message of size m to every processor.

7. **Transportation Primitive.** It performs many-to-many personalized communication with possibly high variance in message size. Let r be the maximum of outgoing or incoming traffic at any processor The transportation primitive breaks down the communication into two all-to-all communication phases where all the messages sent by any particular processor have uniform message sizes [14]. If $r \geq p^2$, the running time of this operation is equal to two all-to-all communication operations with a maximum message size of $O(\frac{r}{p})$.

8. **Order Maintaining Data Movement.** Consider the following data movement problem, an abstraction of the data movement patterns that we encounter in subtask redistribution.

Initially, processor P_i contains two integers s_i and r_i, and has s_i elements of data such that $\sum_{i=0}^{p-1} s_i = \sum_{i=0}^{p-1} r_i$. Let $s_{max} = max_{i=0}^{p-1} s_i$ and $r_{max} = max_{i=0}^{p-1} r_i$. The objective is to redistribute the data such that processor P_i contains r_i elements. Suppose that each processor has its set of elements stored in an array. We can view the $\sum_{i=0}^{p-1} s_i$ elements as if they are globally sorted based on processor and array indices. For any $i < j$, any element in processor P_i appears earlier in this sorted order than any element in processor P_j. In the order maintaining data movement problem, this global order should be preserved after the distribution of the data.

The algorithm first performs a Parallel Prefix operation on the s_i's to find the position of the elements each processor contains in the global order. Another parallel prefix operation on the r_i's determines the position in the global order of the elements needed by each processor. Using the results of the parallel prefix operations, each processor can figure out the processors to which it should send data and the amount of data to send to each processor. Similarly, each processor can figure out the amount of data it should receive, if any, from each processor. The communication is performed using the transportation primitive. The maximum number of elements sent out by any processor is s_{max}. The maximum number of elements received by any processor is r_{max}.

9. **Non-Order Maintaining Data Movement.** The order maintaining data movement algorithm may generate much more communication than necessary if preserving the global order of elements is not necessary. For example, consider the case where $r_i = s_i$ for $1 \leq i < p-1$ and $r_0 = s_0 + 1$ and $r_{p-1} = s_{p-1} - 1$. The optimal strategy is to transfer the one extra element from P_{p-1} to P_0. However, this algorithm transfers one element from P_i to P_{i-1} for every $1 \leq i < p-1$, generating $(p-1)$ messages.

   For data movements where preserving the order of data is not important, the following modification is done to the algorithm: Every processor retains $min\{s_i, r_i\}$ of its original elements. If $s_i > r_i$, the processor has $(s_i - r_i)$ elements in excess and is labeled a source. Otherwise, the processor needs $(r_i - s_i)$ elements and is labeled a sink. The excessive elements in the source processors and the number of elements needed by the sink processors are ranked separately using two Parallel Prefix operations. The data is transferred from sources to sinks using a strategy similar to the order maintaining data movement algorithm.

   The maximum number of outgoing elements at any processor is $max_{i=0}^{p-1}(s_i - r_i)$, which can be as high as s_{max}. The maximum number of incoming elements at any processor is $max_{i=0}^{p-1}(r_i - s_i)$, which can be as high as r_{max}. Therefore, the worst-case running time of this operation is identical to the order maintaining data movement operation. Nevertheless, the non-order maintaining data movement algorithm is expected to perform better in practice.

3 Concatenated Parallelism

A generic divide and conquer algorithm divides a given task into a number of subtasks which are solved recursively until the size of the subtasks is small enough to be solved directly. Consider a task of size N on p processors, initially distributed such that each processor has $\frac{N}{p}$ elements. Without loss of generality, assume that both N and p are powers of 2. For convenience of presentation, assume that a task of size S is divided into two subtasks S_1 and S_2. We are considering problems with $|S_1| + |S_2| \leq |S|$, where $|S|$ denotes the size of S. At stage i ($0 \leq i < \log N$) of the subdivision, 2^i subtasks are created. Our technique can be extended to problems in which the number of subtasks is more than two or the sum of the sizes of the subtasks is larger than the size of the parent task (e.g. binary space partitions).

There are two conventional approaches to solving a given number of tasks in parallel: *Task parallelism* and *Data parallelism*. We describe each of them below and provide a comparison with *Concatenated parallelism* that we propose in this paper. The benefit of concatenated parallelism, as will be clear from the discussion below, comes from eliminating repeated redistributions of subtasks and providing load balancing. Concatenated Parallelism is decidedly superior to Task Parallelism only when the time required to divide a task is linear in the size of the task or close to linear. However, this restriction is valid for a large variety of practical and useful problems.

In task parallel execution, a different group of processors is allocated for each task and all the tasks are executed in parallel. A task parallel divide and conquer algorithm divides the initial task S, using all the p processors. After the subdivision of S into two subtasks S_1 and S_2, each of these will be allocated to a different subgroup of processors. Two allocation schemes are possible: either the processors are divided into two equal sized subgroups, or processor subdivision is proportional to the sizes of S_1 and S_2. The choice depends on topological considerations for a given architecture. This process is repeated recursively until there are p subtasks, one on each processor. A sequential algorithm is now used to solve the subtasks.

When a task S of size N is divided into two subtasks S_1 and S_2, each processor will have some data of both the subtasks. Moving the subtasks to different processor groups causes redistribution of data, which can be performed using the Transportation Primitive outlined in Section 2. Because the total amount of data on each processor is $O(\frac{N}{p})$, such a redistribution cost is proportional to N on both hypercubes and meshes. If the sequential cost of dividing the task S is $O(N^\alpha)$ for some $\alpha > 1$, this cost dominates the cost of redistribution. In this case, strategies to reduce redistribution cost will not be effective. If $\alpha = 1$, the cost of redistribution can no longer be ignored. Also, the constant involved in the redistribution cost (which requires communication) is typically larger than the constant in the subdivision cost (in which the dominant term is usually due to computation) by an order of magnitude. Due to this reason, the communication cost cannot be ignored in practice for some small values of α greater than 1. Tasks where the subdivision cost is like $O(N \log N)$ can

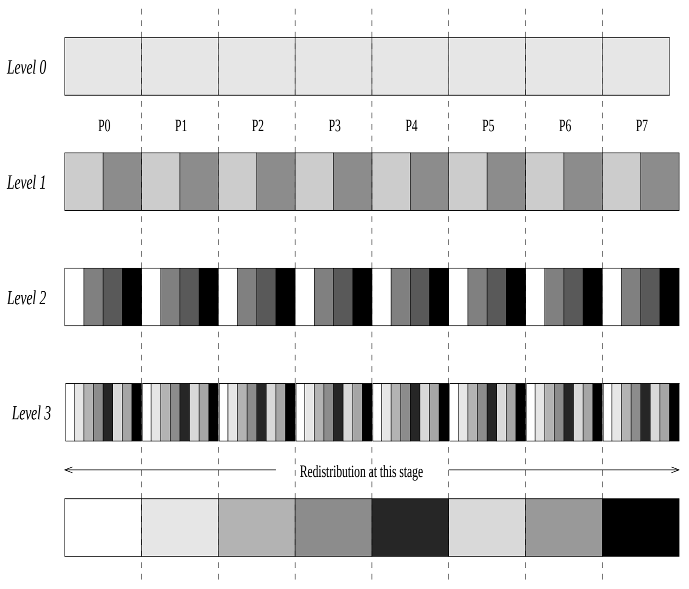

Figure 1: Illustration of Concatenated parallelism for $p = 8$.

potentially benefit by reducing the redistribution cost for practical values of N.

A data parallel algorithm solves one task after another in parallel using all p processors. A task S is subdivided into two and kept locally. Given tasks $S_1, S_2, \ldots S_{2^i}$ distributed on p processors, after i subdivisions, the tasks are further subdivided one after another. Each processor has a portion of each of the subtasks. There are two drawbacks to this approach: The distribution of each subtask across processors may not be uniform. This creates problems due to load imbalance. Also, since the sizes of the subtasks keep decreasing, the "grain-size" of the computation (size of the problem on one processor) keeps reducing. Because unit communication is more expensive than unit computation by two to three orders of magnitude, there is a threshold grain-size below which the communication overhead severely limits any benefits due to parallelization.

In concatenated parallelism, all problems are solved together using all the p processors. A task is divided into subtasks on each processor and subtasks are kept locally. A processor contains portions of each of the 2^i subtasks at the i^{th} level, as shown in Figure 1. All k $(1 \leq k \leq 2^i)$ subtasks are solved together in parallel using all the p processors. This process of dividing a task into subtasks is repeated until a stage is reached where enough subtasks have been created to be distributed to individual processors and solved sequentially. The first such stage is reached at the $\log p^{th}$ level. At this stage there are p subtasks, distributed across all the processors. A redistribution step can gather a subtask on each processor. This works well when subdivision of tasks leads to balanced subtasks. However, when dealing with unbalanced tasks, this can lead to grave load imbalance. This can be rectified by using concatenated parallelism further, continuing to divide the tasks beyond $\log p$ levels. This helps to achieve a better load balance as relatively fine-grained tasks are being gathered for allocation to processors. Apart from load balancing considerations, the level at which

Algorithm 1 *Concatenated parallel algorithm*

```
$N$ : Total size of the task.
$p$ : Total number of processors labeled from 0 to $p-1$.
$f(s,p)$ : Computation work for a task of size $sp$ distributed on $p$ processors.
$g(s,p)$ : Communication required along with performing $f(s,p)$.
$h(s,p)$ : Redistribution cost for tasks with total size $s$ on $p$ processors.
$Q_j$ : size of the task $j$, $1 \leq j \leq 2^k$, at stage $k$. Initially $k$=0 and $|Q_1| = N$.
$x$ : $1 \leq x < N$, a task of size $N$ splits into subproblems of size $x$ and $N-x$.
$K$ : A value for $|Q_j|$ below which partitioning will stop.
done = false
while(! done)
    while ($\sum_j g(Q_j, p) < h(\sum_j Q_j, p)$ and $\max_j |Q_j| > K$)
        Step 1. Split each of the $j$ tasks into two subtasks by appropriate subdivision
                of $Q_j$ on all the processors. Increment $k$ by 1.
    /*Redistribution phase */
    If $\max_j |Q_j| < K$ then
        Step 2. Redistribute the task $Q_j$, $1 \leq j \leq 2^k$ to individual processors.
        Step 3. done = true
    else
        Step 4. Redistribute the problems $Q_j$, $1 \leq j \leq 2^k$ to processor pools of size
                $\frac{P}{2^m}$ for a maximum $m$, $1 \leq m \leq \log P$ such that $\sum_j g(Q_j, \frac{p}{2^m}) <$
                $h(\sum_j Q_j, \frac{p}{2^m})$.
        Step 5. Set $p$ to $\frac{p}{2^m}$.
        Step 6. If ($m == \log P$) done = true.
```

Figure 2: Concatenated parallel algorithm

concatenated parallelism should be stopped and redistribution done depends on the comparative cost of task subdivision versus task redistribution. A later section describes redistribution strategies and criteria for applying redistribution. The framework for Concatenated Parallelism is illustrated in Figure 2.

At every stage, a processor works with $\frac{N}{p}$ elements even though they might belong to different subtasks. This is an important feature of the algorithm because it guarantees load balance at every stage even though individual subtasks are not balanced. Another view of the Concatenated Parallelism is that the grain-size is always $\frac{N}{p}$ and never decreases irrespective of how small the individual subtasks are. Divide and conquer algorithms resulting in unbalanced or randomized subdivisions especially benefit from this algorithm.

We distinguish between three different types of divide and conquer trees: deterministic balanced trees, deterministic unbalanced trees and randomized trees. A generic divide and conquer algorithm for Task Parallelism is modeled by the recurrence relation which can be written as $T(N,p) = \max\{T(x, \lfloor \alpha p \rfloor), T(N-x, \lceil (1-\alpha)p \rceil)\} + f(N,p) + g(N,p)$, where α is a factor governing processor allocation, $f(N,p)$ is the computation cost and $g(N,p)$ is the communication cost in dividing a task of size N on p processors. A generic recurrence relation for Data Parallelism is $T(N,p) = T(x,p) + T(N-x,p) + f(N,p) + g(N,p)$. In Concatenated parallelism all tasks at each stage are solved together spooling the communication, unlike the case of Data Parallelism, where tasks are solved one after another. As discussed previously, we only consider problems for which $f(N,p) = O(\frac{N}{p})$. $g(N,p)$ is chosen to be $O((t_s + t_w)\log p)$ in our analysis on both hypercubes and meshes, because the communication required for subdividing a task typically requires a combination of Parallel Prefix, Combine and Broadcast operations for many applications. Of course, given a particular problem, analysis specific to that problem can always be performed.

In a deterministic balanced divide and conquer algorithm, problems are split into two halves, and therefore x is $\frac{N}{2}$. In the other two categories, x has no guaranteed size. In deterministic unbalanced trees, x can be any integer from 1 to N. However, the value of x is completely determined by the input and two runs on the same input will give the same value of x. In randomized trees, x can take any value from 1 to N with some probability associated with each possible value. Two runs on the same input may lead to two different values of x. In studying Task Parallelism, we either allocate half the processors to each subproblem in which case α will be set to $\frac{1}{2}$, or allocate processors proportional to the subtask sizes in which case α will be set to $\frac{x}{N}$.

At a stage i, in Concatenated Parallelism, each of the 2^i subtasks need to be divided into two subtasks. This requires communication between processors which can be spooled together. Since a processor contains portions of all tasks, it might need to broadcast data for a subtask. Different processors might end up broadcasting elements for different subtasks. To avoid 2^i broadcasts, we adopt the following strategy for spooling communication: Each processor has an array of size 2^i corresponding to the 2^i subtasks, with all elements initialized to zero. If a processor has the element to broadcast for a subtask, it fills the corresponding element of the array with that element. By doing a Combine operation on this array using the '+' operation, the required elements for all subtasks are stored on each processor.

Our goal is to analyze Concatenated Parallelism and compare it with both Task Parallelism and Data Parallelism. One can easily show that Data Parallelism always performs worse than Concatenated Parallelism irrespective of the nature of the tree, size of the problem or number of processors. Consider a case when the divide and conquer tree is evaluated up to i levels and let the 2^i subtasks be $S_1, S_2, \ldots S_{2^i}$. Let ${S_j}^k$ refer to the portion of the j^{th} subtask on the k^{th} processor, $0 \leq k < p$ and $1 \leq i \leq 2^i$. Let T_{dp} denote the computation time to divide all these subtasks into further subtasks. Since tasks are performed in sequence and each subsequent task is solved on all p processors, the computation time for subtask j is $\max_{k=0}^{p-1} |{S_j}^k|$. The time for all the 2^i tasks is $\sum_{j=1}^{2^i} \max_{k=0}^{p-1} |{S_j}^k|$. It is easily seen that $T_{dp} \geq O(\frac{N}{p})$. The equality is obtained when subtasks are uniformly distributed among the processors. In Concatenated Parallelism, the computation time is $O(\frac{N}{p})$. Also, in Concatenated Parallelism, the time spent in necessary communication for subdividing the subtasks is always smaller than communication time spent in Data Parallelism. This is because communication for all the subtasks is spooled together and this saves expensive set-up costs in individual communications. One can easily see that even when the sequential cost of subdividing a task of size N is any general function of N (not necessarily linear), Concatenated Parallelism always provides at least as good a load balance as Data Parallelism ($\max_{k=0}^{p-1} \sum_{j=1}^{2^i} (|{S_j}^k|)^\alpha \leq \sum_{j=1}^{2^i} \max_{k=0}^{p-1} (|{S_j}^k|)^\alpha$). Therefore, we limit our theoretical and experimental comparisons to comparing Concatenated Parallelism with Task Parallelism only.

The following sections present each of the three categories of divide and conquer algorithms. For each category, we analyze Concatenated Parallelism and Task Parallelism with the assumption that $f(N,p) = O(\frac{N}{p})$ and $g(N,p) = O((t_s + t_w) \log p)$. A different $g(N,p)$ can be analyzed for a problem if the need arises. This is done in a later section for construction of multidimensional binary search trees, where this function is $O((t_s + t_w) \log p \log N)$. Example applications follow each category and performance results on the CM-5 are provided to supplement the analysis.

4 Deterministic Balanced Parallel Divide and Conquer

Deterministic balanced divide and conquer algorithms result in the subdivision of a task of size N into two subtasks of size $\frac{N}{2}$, at each stage. Thus after i iterations, each of the 2^i subtasks will have size $\frac{N}{2^i}$. Every processor contains portions of each subtask. A balanced divide and conquer algorithm results in p perfectly balanced tasks after $\log p$ iterations. An appropriate criteria is used for redistributing the p subtasks to individual processors. At a stage i, $\sum_{k=0}^{2^i} f(\frac{N}{2^i}, p)$ is the computation cost to achieve the subdivision of all tasks. Communication is combined for all the tasks. After $\log p$ steps of the subdivision, a final redistribution cost $h(N,p)$ is incurred.

The running time to create p subtasks, one for each processor, using spooling of communication for deterministic balanced divide and conquer is $\sum_{i=0}^{\log p-1} O(2^i \frac{N}{2^i p} + t_s \log p + t_w 2^i \log p)$ on both a hypercube and a mesh. Adding $h(N,p) = O(t_s p + t_w \frac{N}{p})$ on a hypercube and $O(t_s \sqrt{p} + t_w \frac{N}{\sqrt{p}})$ on a mesh, we obtain $O(\frac{N}{p} \log p + t_s(p + \log^2 p) + t_w \frac{N}{p})$ as the running time on a hypercube and $O(\frac{N}{p} \log p + t_s(\sqrt{p} + \log^2 p) + t_w \frac{N}{\sqrt{p}})$ on a mesh, for a deterministic balanced divide and conquer algorithm.

In a task parallel approach which divides a processor subgroup equally at each level, there are 2^i subtasks, each being distributed on processor subgroups of size $\frac{p}{2^i}$ at some level i. Each subtask has size $\frac{N}{2^i}$ which needs to be solved in parallel on the processor subgroup. We obtain the running time for this method using the recurrence relation for Task Parallelism as $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i} / \frac{p}{2^i} + t_s(\frac{p}{2^i} + \log \frac{p}{2^i}) + t_w(\log \frac{p}{2^i} + \frac{N}{p}))$ on a hypercube. The redistribution cost is a part of the recurrence relation in this case. This gives the running time as $O(\frac{N}{p} \log p + t_s(p + \log^2 p) + t_w \frac{N}{p} \log p)$. The corresponding running time on a mesh is $\sum_{i=0}^{\log p-1} O(\frac{N}{2^i} / \frac{p}{2^i} + t_s(\sqrt{\frac{p}{2^i}} + \log \frac{p}{2^i}) + t_w(\log \frac{p}{2^i} + \frac{N}{p}\sqrt{\frac{p}{2^i}}))$ $= O(\frac{N}{p} \log p + t_s(\sqrt{p} + \log^2 p) + t_w \frac{N}{\sqrt{p}})$.

Note that this category yields the best case for task parallelism. However, concatenated parallelism does better in this case since it reduces the redistribution costs from $\frac{N}{p} \log p$ to $\frac{N}{p}$ on a hypercube.

4.1 Applications - Multidimensional Binary Search trees

Construction of multidimensional binary search trees (abbreviated k-d trees) [1], in which dividing a task is based on the median element of the corresponding data set, uses a balanced divide and conquer algorithm. The root of the k-d tree corresponds to the set of all points. Assume k dimensional data with $d_1, d_2, \ldots, d_k$ as the dimensions. Choose a dimension d_l and partition points into two sets, one containing points with coordinates less than or equal to this median along dimension d_l and another containing points with coordinates greater than the median. The two partitions are represented by the children of the root node. The tree is built recursively until each node corresponds to a prespecified number of points.

Consider the construction of a k-d tree of N points on p processors with $\frac{N}{p}$ points on each processor initially. We use a randomized median finding algorithm [2] to calculate the median for points along dimension d_1. Points are divided into two subsets, S_1 and S_2, using the median. S_1 contains points with their d_1 coordinate values less than or equal to the median and S_2 contains points having their d_1 coordinate values greater than the median. Considering the processors partitioned into two halves, task parallelism would gather S_1 on the lower half subgroup of processors and S_2 on the upper half subgroup of processors. This process is repeated recursively, changing dimensions in a cyclic manner to find the median, until each processor has a subset and each processor subgroup contains a single processor. A sequential algorithm is then applied to solve the problem locally.

Thus, for k-d tree construction, $f(N,p)$ is the cost of parallel median finding of N elements on p processors plus the cost of partitioning local data into two portions, one containing elements less than or equal to the median and another containing elements greater than the median. $g(N,p)$ is the associated communication cost. For this method, $f(N,p)$ is $O(\frac{N}{p})$ and $g(N,p)$ is $O((t_s + t_w)\log p \log N)$. Using Task Parallelism, building the first $\log p$ levels of the tree on a hypercube requires $\sum_{i=0}^{\log p - 1} O(\frac{N}{2^i}/\frac{p}{2^i} + (t_s + t_w)\log\frac{p}{2^i}\log\frac{N}{2^i} + k\frac{N}{p} + t_s\frac{p}{2^i} + t_w\frac{kN}{p}) = O(\frac{kN}{p}\log p + t_s(p + \log^2 p \log N) + t_w(k\frac{N}{p}\log p + \log^2 p \log N))$ time. The time required on a mesh is $O(\frac{kN}{p}\log p + t_s(\sqrt{p} + \log^2 p \log N) + t_w(\frac{kN}{\sqrt{p}} + \log^2 p \log N))$.

A Concatenated Parallel algorithm divides the tasks across each processor. Starting with a single task, subdivision is applied recursively to each subset resulting in 2^i subsets after i partitions. After redistribution, a sequential algorithm is used on each processor to construct the k-d tree locally. In this case, communication for the 2^i subtasks is combined together using the technique described in an earlier section. The cost of median finding at level i is $O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N)$ for a hypercube and $O(\frac{N}{p} + t_s \log p \log N + t_w 2^i \log p \log N)$ for a mesh. Median finding is done for each partition and the data in a partition is automatically subpartitioned into two subsets in the process. Cost of the redistribution is $O(\frac{kN}{p} + t_s p + t_w \frac{kN}{p})$ on a hypercube and $O(\frac{kN}{p} + t_s\sqrt{p} + t_w\frac{kN}{\sqrt{p}})$ on a mesh. Combining these costs, the running time for a deterministic balanced divide and conquer algorithm using concatenated parallelism is $O(\frac{N}{p}(\log p + k) + t_s(p + \log^2 p \log N) + t_w(\frac{kN}{p} + p\log p \log N))$ on a hypercube. The corresponding time on a mesh is $O(\frac{N}{p}(\log p + k) + t_s(\sqrt{p} + \log^2 p \log N) + t_w(\frac{kN}{\sqrt{p}} + p \log p \log N))$.

The data movement due to redistribution reduces by a factor of $\log p$ on hypercubes by using Concatenated Parallelism. The cost of redistribution does not reduce on the mesh. Computation cost reduces from $O(\frac{N}{p}k\log p)$ to $O(\frac{N}{p}(k + \log p))$ when Concatenated Parallelism is used. Therefore, higher dimensional data results in even better performance by using Concatenated parallelism.

4.2 Experimental Results

We have implemented the construction of two dimensional binary search trees on the CM-5. Two algorithms are compared: Task parallelism with processor groups being divided equally into two subgroups at each stage and Concatenated parallel. Figure 3 presents a comparison of the two approaches for various values of $\frac{N}{p}$ for tree construction up to $\log p$ levels. The results show that concatenated parallelism works better than task parallelism for balanced divide and conquer only when there are a large number of points on a processor. The gains from reducing redistribution cost can be obtained only when the data is large in the balanced case. In the unbalanced case, as we will observe in a later section, come both from reducing redistribution cost and providing good load balance. For higher dimensional data, the cost of redistribution is more significant since it involves

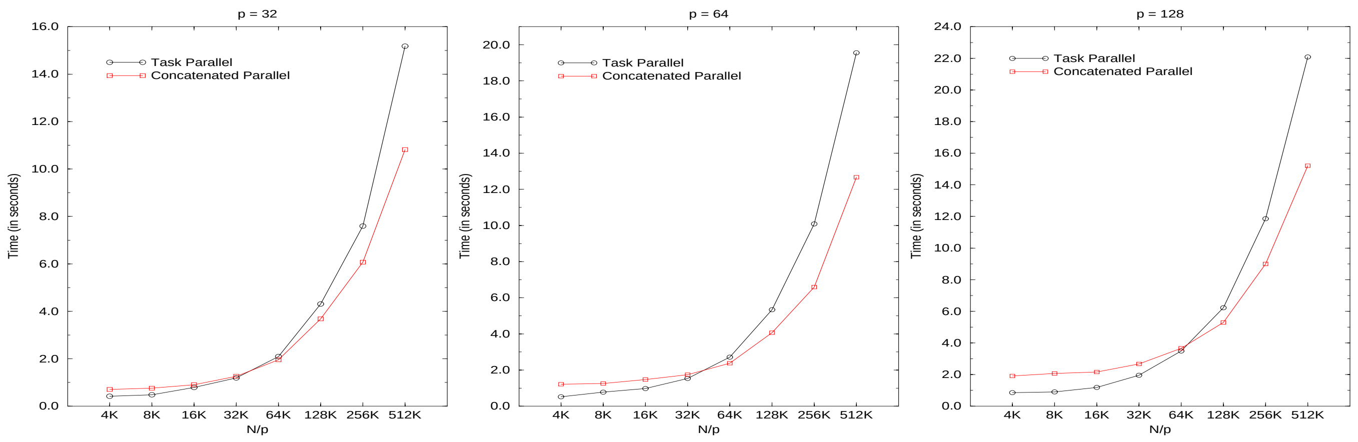

Figure 3: Comparison of Task Parallelism and Concatenated Parallelism for balanced divide and conquer in construction of k-d trees for different values of N/p.

a higher volume of data to be redistributed and hence Concatenated Parallelism is expected to perform even better.

Figure 4 shows the component times for k-d tree construction up to $\log p$ levels with $N = 2\text{M}$ and 4M. The total time for tree construction is divided into computation time and communication time for median finding at all levels and data redistribution time. We observe that redistribution time for Concatenated Parallelism is significantly lower than for Task Parallelism. The computation time for Concatenated Parallelism is slightly higher due to added list management. Concatenated Parallelism involves all p processors at all stages for communication whereas for Task Parallelism communication occurs in processor subgroups. Certain parallel machines (e.g CM-5) use a special network when all the processors are participating in the communication. The cost of global communication is lower in such a case.

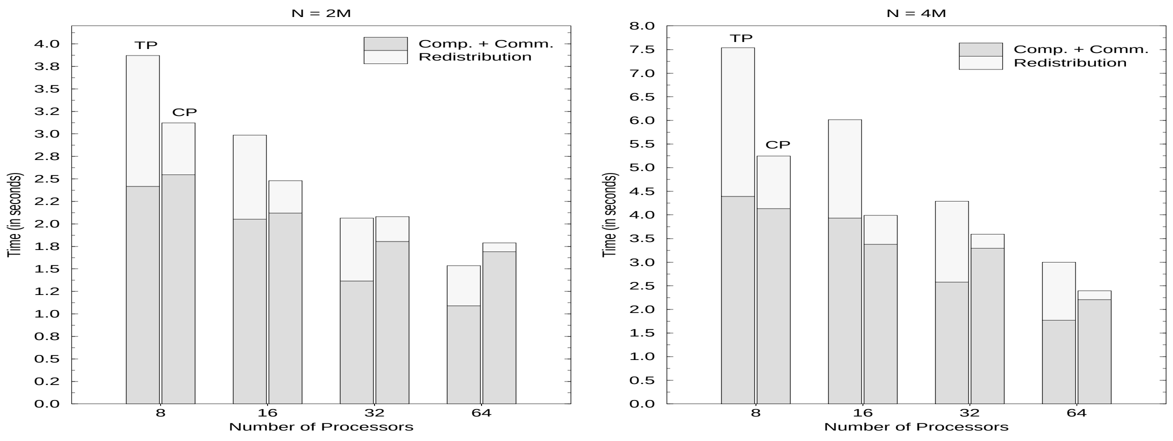

Figure 4: Time for tree construction up to $\log p$ levels divided into computation time for splitting the points plus the associated communication time and cost of redistribution for $N = 2\text{M}$ and 4M on 8, 16, 32 and 64 processors. (TP: Task Parallel, CP: Concatenated Parallel)

5 Deterministic Unbalanced Parallel Divide and Conquer

Subdivision of tasks in certain divide and conquer algorithms result in subtasks of different sizes. The subdivision at each level remains unchanged for a given input and hence the algorithms are deterministic in nature. Using Concatenated parallelism, the work associated with the portions of subtasks on a processor may vary as a result of such subdivisions, but each processor still works on $\frac{N}{p}$ elements. This method provides good load balancing for unbalanced parallel divide and conquer.

The number of points belonging to a subset may be highly imbalanced. Consider a problem of size N at some stage of the subdivision process. Suppose the problem is divided into two subproblems of sizes c and $N-c$, where c is a constant. Note that this definition will not subdivide a problem of size c further. This will lead to the worst-case analysis as the problem size is reduced by a constant. There is only one task to be solved at each stage. Concatenated Parallelism works the same as Data Parallelism in this worst case. A task is subdivided into two tasks at each stage. Let K be the size of a task after which it would not be subdivided. Subdivision of tasks is continued till a stage where all tasks have a size at least K. The value of K is set appropriately so that there are at least p subtasks and these can be allocated to processors to provide adequate load balance. A choice of $K = \frac{N}{p}$ is good because it guarantees that no processor will have more than $\frac{2N}{p}$ elements. At a level i, all the p processors are working on a problem of size $N - ic$. Choosing values of $f(N,p)$ as $O(\frac{N}{p})$ and $g(N,p)$ as $O((t_s + t_w)\log p)$ the running time of a Concatenated Parallel unbalanced divide and conquer is $O(\frac{N^2}{p^2}) + \sum_{i=0}^{\lceil\frac{N-K}{c}\rceil-1} \frac{N-ic}{p} + (t_s + t_w)\log p$. Redistribution cost is $O(t_s p + t_w \frac{N}{p})$ on a hypercube and $O(t_s p + t_w \frac{N}{\sqrt{p}})$ on a mesh. The time for unbalanced divide and conquer using Concatenated Parallelism is $O(\frac{N^2}{p} + t_s(p + \frac{N-K}{c}\log p) + t_w(\frac{N-K}{c}\log p + \frac{N}{p}))$ on a hypercube and $O(\frac{N^2}{p} + t_s(p + \frac{N-K}{c}\log p) + t_w(\frac{N-K}{c}\log p + \frac{N}{\sqrt{p}}))$ on a mesh.

Task Parallelism can either partition the processors into two equal subgroups and allocate each to a subtask, or processor allocation to subgroups can be done proportional to subtask sizes. Unbalanced subdivisions lead to idling of processors as some processors have more work to do than others in the case where half of the processors are allocated to a subgroup. The earlier the imbalance occurs while subdividing, the worse the performance will be because larger subgroups of processors are allocated during the earlier levels. Assume that a problem of size N is subdivided into subproblems of sizes c and $N-c$, represented by the following recurrence relation $T(N,p) = T(N-c, \frac{p}{2}) + f(N,p) + g(N,p) + h(N,p)$. The redistribution cost at each level can be bounded by an all-to-all communication using the transportation primitive. Substituting the generic values for $f(N,p)$ and $g(N,p)$ defined earlier, we obtain a running time to create p subtasks as $\sum_{i=0}^{\log p-1}(N - ic)/\frac{p}{2^i} + (t_s + t_w)\log\frac{p}{2^i} + t_s\frac{p}{2^i} + t_w(N-ic)/\frac{p}{2^i}$ on a hypercube and $\sum_{i=0}^{\log p-1}(N-ic)/\frac{p}{2^i} + (t_s + t_w)\log\frac{p}{2^i} + t_s\sqrt{\frac{p}{2^i}} + t_w(N-ic)/\sqrt{\frac{p}{2^i}}$ on a mesh. After $\log p$ iterations, one processor contains a task of size $O(N - c\log p)$. The running time for deterministic unbalanced divide and conquer algorithms using Task Parallelism is then $O((N - c\log p)^2 + t_s(p + \log^2 p) + t_w N)$ on a hypercube and $O((N - c\log p)^2 + t_s(\sqrt{p} + \log^2 p) + t_w N)$ on a mesh.

The computation cost using Task Parallelism is $O((N - c\log p)^2)$. This is reduced to $O(\frac{N^2}{p})$ using Concatenated Parallelism. Processor partitioning proportional to subproblem sizes for Task Parallelism would lead to allocating a processor to a subproblem of size c. This would result in $O((N - cp)^2)$ computation, not much better than the other case of Task Parallelism. Concatenated Parallelism provides better load balance than both kinds of Task Parallelism. Redistribution cost is reduced from $O(N)$ to $O(\frac{N}{p})$ on a hypercube, and to $O(\frac{N}{\sqrt{p}})$ on a mesh by using Concatenated Parallelism.

Algorithms for building quadtrees and finding the convex hull using the quickhull technique follow the unbalanced divide and conquer paradigm. We describe both these algorithms below.

5.1 Applications - Quadtrees and QuickHull

5.1.1 Quadtrees

Consider building a quadtree for a set of points S in a $R \times R$ planar space [15]. Four partitions each of size $\frac{R}{2} \times \frac{R}{2}$ are created using the median coordinates of the space. This process is repeated recursively until each point in the set belongs to a separate partition. At each stage of the subdivision the number of points belonging to each partition depends on the input data. It can potentially lead to unbalanced partitions and hence an unbalanced tree.

Given N points and p processors with $\frac{N}{p}$ points on each processor initially, each processor partitions its points into four subsets, each of the subsets corresponding to points lying in a quadrant of the given sample space. Each subset corresponds to a child node in the tree rooted with a node representing a subspace with dimensions $R \times R$ for some $m > 0$. Each of these subsets is partitioned recursively giving rise to 4^i subsets after the i^{th} stage of the subdivision. An appropriate stage is chosen to redistribute subsets to individual processors to be solved sequentially.

5.2 QuickHull

In computational geometry, algorithms for finding the convex hull of points in space [12], like QuickHull, follow the divide and conquer approach of quicksort. In this case the subpartitions at any stage of partitioning are not guaranteed to be balanced as they depend on the input data. It follows that, given n points, the sequential algorithm for quickhull takes $O(n^2)$ worst case time, the average case run time being $O(n \log n)$.

The convex hull of a set S containing N points is the smallest convex set containing S. It is represented by a polygonal chain [1] which contains the points on the convex hull. The Quickhull algorithm partitions S into two subsets, each of which computes a polygonal chain whose concatenation gives the convex hull polygon. The initial partition is determined by the line passing through l and r, the two points with the minimum and the maximum x-coordinates. Let $S^{(1)}$ be the subset of the points on or above the line lr and let $S^{(2)}$ be the subset of points below lr. A subset $S^{(k)}$ is processed in the following manner: A point $h \in S^{(k)}$ is determined such that the triangle (hlr) has the maximum area among all the triangles (plr), $p \in S^{(k)}$. h is a point on the convex hull. The next step is to construct two lines, one directed from l to h, $(\overline{lh})$, and another directed from h to r, $(\overline{hr})$. Points belonging to $S^{(k)}$ are tested with respect to these two lines. Clearly, points in the triangle (lrh) are interior points and have to be discarded. Points not to the left of $\overline{hr}$ but lying on or to the left of $\overline{lh}$ form a set $S^{(k,1)}$. $S^{(k,2)}$ is similarly formed by the points not to the left of $\overline{lh}$ but on or to the left of $\overline{hr}$. This process is repeated recursively on the newly formed subsets. A parallel algorithm for quickhull will start by finding the points with minimum and maximum x coordinates on all processors and performing a Combine operation to find the global minimum, min_x,and the global maximum max_x. These two points are on the hull as they are at extremities of the set. Each processor finds the point h, that gives the maximum area triangle with the line passing through min_x and max_x. Another Combine operation finds the maximum, h_{max}, among all the h points. h_{max} is a point on the hull. Points on the right of $\overline{h_{max} max_x}$ and to the left of $\overline{min_x h_{max}}$ form a subpartition. Another subpartition is defined for the points on the right of $\overline{min_x h_{max}}$ but lying to the left of $\overline{h_{max} max_x}$. Each of these subsets is solved in parallel recursively on all p processors, creating 2^i subsets after i steps of the algorithm. An appropriate stage is chosen for redistribution of subsets to individual processors, which apply the sequential algorithm to each subset separately.

5.3 Experimental Results

We present results for the quickhull algorithm on the CM-5. A concatenated parallel quickhull is compared with a task parallel quickhull implementation in Figure 5. The convex hull is constructed for 128K, 512K and 2M random points on 4, 8, 16, 32, 64 and 128 processors. Concatenated parallel algorithm clearly performs much better than a task parallel quickhull. We observe from the results that the gains of Concatenated Parallelism over Task Parallelism are diminishing with the increase in p. A smaller value of $\frac{N}{p}$ results in lesser load imbalance for Task Parallelism, a factor where Concatenated Parallelism gains the most. Redistribution costs are also lower and the gains from lowering this cost are also lower. These observations are endorsed by the analysis of both the methods in the previous section.

Similar results on experiments with building of quadtrees lead us to the conclusion that Concatenated Parallelism parallelizes well for deterministic unbalanced divide and conquer methods.

[1]A *chain* is a planar straight-line graph with vertex set $u_1, u_2, \ldots, u_p$ and edge set (u_i, u_{i+1}): $(1 \leq i \leq p-1)$

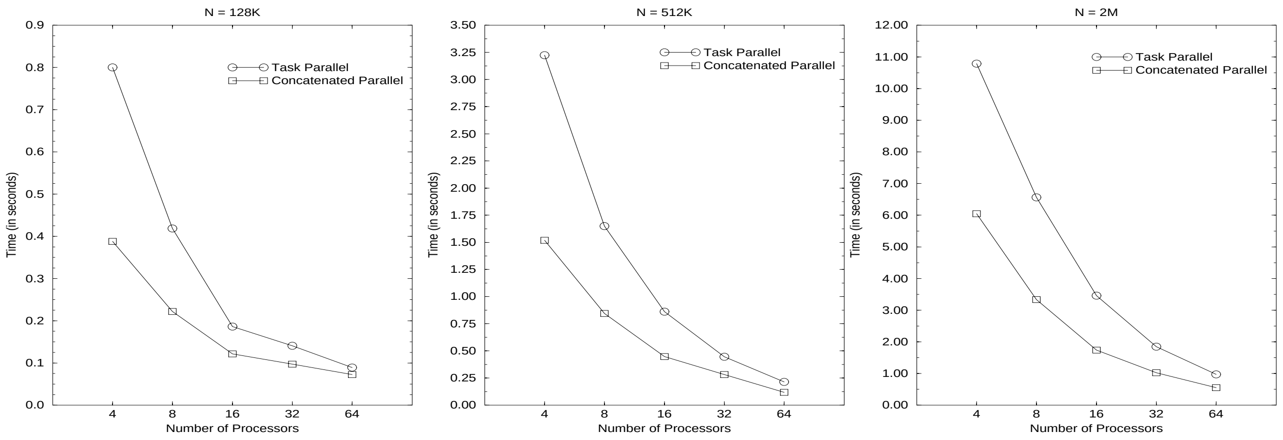

Figure 5: Comparison of Task Parallelism and Concatenated Parallelism for unbalanced divide and conquer in quickhull for N = 128K, 512K and 2M random data.

6 Randomized Parallel Divide and Conquer

The sizes of tasks after subdivision into two subtasks can be considered as random variables in some divide and conquer algorithms. Randomized quicksort is such an application. The subdivision process does not guarantee balanced partitions. However, something can be said about the expected value of the subtasks and hence about the number of iterations required to reduce the problem size to a specified level so that redistribution can be done. In this case the subtask sizes are a random variables. For the same input we could get different subtask sizes, because the criteria for subdivision is random and each element in the set is equally likely to be picked. In cases of unbalanced partitioning, Concatenated Parallelism will perform better as seen in the previous section.

In a randomized divide and conquer tree, the sizes of the subtasks of a given task depend on random choices made in the algorithm. We again limit our attention to the case where each task divides into two subtasks and sum of the sizes of the subtasks is the same as the size of the parent task. Suppose a task of size N is split into two subtasks. The sizes of the subtasks are given by x and $N - x$, where x is a random variable that can take any value between 1 and N. There is a probability associated with x assuming each of the allowable values. The probability distribution is often uniform. For example in quicksort, a random element from a given array is picked as a pivot and used to partition the array into two subarrays.

For randomized divide and conquer trees with uniform distributions and for which the cost of dividing a set is linear in the size of the set, a sequential analysis similar to quicksort shows that the expected running time is $O(N \log N)$, even though the worst-case run time is $O(N^2)$. However, there is a severe drawback to using Task Parallelism with equal subdivision of processors for randomized trees. Suppose that the subtask sizes when the root of the tree is subdivided are extremely unbalanced. This does not affect the expected running time of the subtasks in the sequential algorithm. However, half the processors are committed to a small subtask in Task Parallelism and the effect of this allocation will have a significant effect on the running times of all the subtasks in the subtree of the larger subtask.

For uniform distributions, the expected size of a subtask at level i of the tree is $\frac{N}{2^i}$. However, the variance in the sizes of the subtasks at level i of the tree decreases with increase in i. When a task of size N is split into two subtasks, the variance in the sizes of the subtasks can be shown to be $O(N)$, of the same order as the expected size. Concatenated Parallelism exploits this by advancing on the tree, level by level in parallel using all the processors until a balanced distribution of the subtasks to individual processors is possible. Since the variance decreases with levels, one can expect the performance of Concatenated Parallelism on randomized trees with uniform distribution to be similar to its performance on deterministic balanced trees.

6.1 Applications - Quicksort

Consider sorting a set S of N numbers in ascending order. Pick an element x from S and treat it as a pivot to partition S into two subsets S_1. containing elements smaller than or equal to x, and S_2, containing the remaining elements. This process is recursively applied to S_1 and S_2 to get the sorted order for S.

Assume a set S containing N elements divided equally among p processors. To parallelize quicksort on p processors assume that each processor contains $\frac{N}{p}$ elements to begin with. Pick a pivot at random from any of the p processors and broadcast it to all others. Each processor subdivides their elements using this pivot. This process is repeated recursively for each of the two subsets of elements. After i such subdivisions there are 2^i subsets to work with. An appropriate stage is chosen to redistribute these subsets to processors which apply the sequential quicksort algorithm to the sets allocated on them.

6.2 Experimental Results

Figure 6 presents a comparison between a task parallel and a concatenated parallel implementation of quicksort. We report results of experiments on sorting 128K, 512K and 2M random floating point numbers using 4, 8, 16, 32 and 64 processors. Task Parallelism with processor allocation proportional to the problem size [Task Parallelism (B)] performs better than Task Parallelism where half the processors are allocated to each subproblem [Task Parallelism (A)]. Approach (A) has higher imbalance due to which local sorting becomes a significant factor since some processors might have many more elements than others. Concatenated Parallelism has a lower running time than both the other methods. Randomized partitioning strategies can result in unbalanced partitions at each level leading to load imbalance and idling of processors. Load balancing is one advantage that

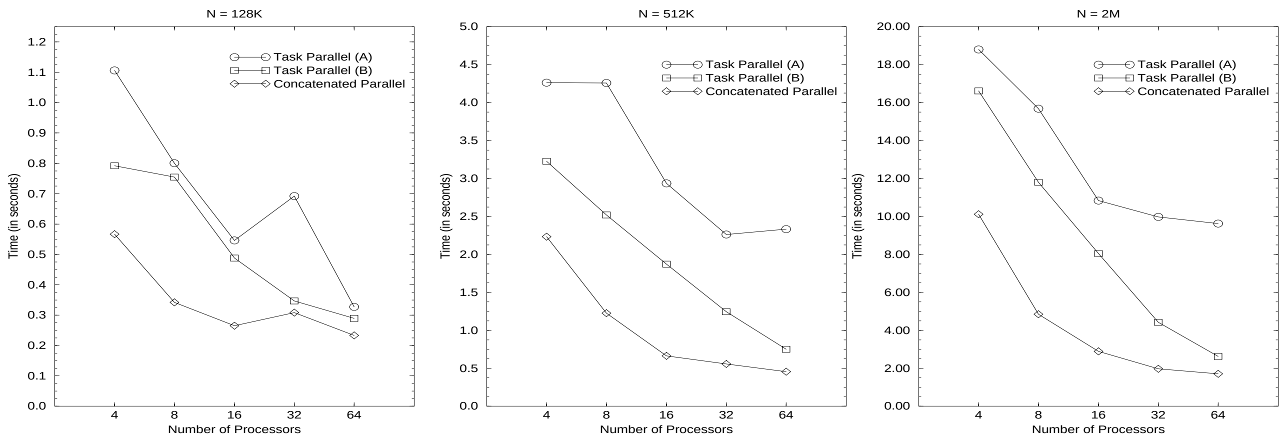

Figure 6: Comparison of Task Parallelism allocating half the processors to a subproblem (A), Task Parallelism with processor allocation proportional to subproblem sizes (B) and Concatenated Parallelism for randomized divide and conquer in quicksort for N = 128K, 512K and 2M random data.

Concatenated Parallelism offers to parallelize randomized divide and conquer methods. For Task Parallelism, processor allocation proportional to subproblem sizes may not always be possible due to topological considerations, or whenever possible may have higher overheads.

7 Redistribution Strategies

Subdivision of tasks into subtasks should be stopped as soon as the remaining subtasks can be distributed to individual processors. Redistribution of subtasks to processors occurs at a level when there are at least p subtasks and the size of each has reached a prespecified value so that they can be distributed to processors to be solved sequentially. This constant value determines the "grain size" of the largest task that we have in a pool of tasks ready for redistribution. Redistribution can potentially be done when the cost of subdividing a problem is more than the cost of redistribution. Consider an example where $\frac{\log p}{2}$ iterations of the subdivision have been performed and $\sqrt{p}$ subtasks created. If it is the case that partitioning costs at this stage are higher than redistribution costs then the $\sqrt{p}$ tasks can be redistributed to subpools of processors each of size $\frac{p}{2^m}$, $0 < m \leq \log p$. However, redistribution should only be done when each subgroup of processors can be allocated approximately equal work.

Using $\frac{N}{p}$ as the size of the largest subtask before redistribution this will ensure that no processor gets tasks whose sizes sum up to more than $\frac{2N}{p}$. A load balancing algorithm is used to allocate the subtasks to processors such that each processor gets nearly equal amount of work. There are various load balancing strategies that can be used in this case. We describe two techniques below which are illustrated in Figure 7.

1. **Order preserving combinations** – At a stage k, when redistribution is to be performed, order the list of 2^k (0 to $2^k - 1$) subtasks by using their indices in the list. The average work associated with a task on a processor should be $\frac{2^m N}{p}, 0 < m \leq \log p$, for a processor pool of size $\frac{p}{2^m}$. Beginning with the leftmost task entry in the ordered list, we can scan the task list from left to right allocating a task to a processor(pool) until the total work allocated to a processor (pool) is no more than the average value. However, this technique does not lead to the optimal load balance because we can only guarantee that the total work on a processor after the redistribution will not be more than twice the average work. This technique preserves ordering of the subproblems which might be essential in some applications where data ordering is important.

2. **Relative order combinations** – An idea similar to the one used for modified order maintaining load balance [2] can be used here. At stage k, the 2^k subtasks are sorted in increasing order by the work associated with them. Task allocation to a processor is guided by two pointers, one placed at the start and another at the end of the sorted list. Problems are combined by moving the left pointer to the right and the right pointer to the left, allocating tasks to processors till each processor contains at least the average work. This strategy helps in allocating tasks to processors with total size on each processor as close to average as possible. Larger tasks are combined with small tasks and towards the end smaller tasks are aggregated which would help in not exceeding the average value by much. However, it destroys the ordering of the subtasks which might be of importance in some applications. If this happens at an intermediate level of the partitioning the ordering of tasks can be regained by another redistribution step.

8 Conclusions

In this paper, we have proposed a new strategy called *Concatenated Parallelism* to efficiently parallelize applications resulting in divide and conquer trees. We compare this strategy to the two traditional approaches used in solving such problems – *Task Parallelism* and *Data Parallelism.* Task Parallelism causes significant redistribution of data at every level of the divide and conquer tree. However, it has the advantage that subtasks are uniformly distributed and allocated to smaller groups of processors. Data Parallelism avoids redistribution of data. But, it causes load imbalance and solves smaller sized subtasks using all processors, thus reducing practical efficiency. Concatenated Parallelism attempts to combine the advantages of both the approaches. It avoids redistribution of data, and by combining the computation and communication of the subtasks, avoids load imbalance and grain-size problems. The minimum grain-size $\frac{N}{p}$, required for effective parallelization of an algorithm typically increases with the value of p. Data Parallelism decreases the grain-size and keeps the number of processors fixed. Concatenate Parallelism maintains the

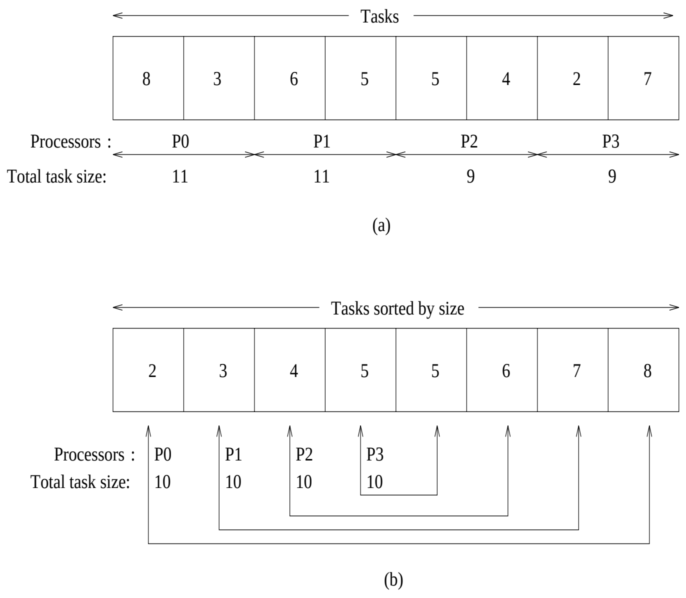

Figure 7: Illustration of load balancing during redistribution of tasks (a) Order preserving (b) Relative order combinations. Task allocation to processors is more balanced in (b).

grain-size and keeps the number of processors fixed. However, Task Parallelism maintains the grain-size and decreases the number of processors, thus making the grain-size increasingly more effective. This is the only advantage of Task Parallelism over Concatenated Parallelism.

Concatenated Parallelism always yields better results than Data Parallelism irrespective of the nature of the divide and conquer tree. This is true irrespective of any parameter including the number of children per node, amount of work involved in dividing a task and the distribution of the sizes of the subtasks. It also holds true irrespective of the divide and conquer tree being balanced, unbalanced or randomized.

Concatenated Parallelism can outperform Task Parallelism only when the cost of redistribution of data is significant when compared to the cost of dividing the subtasks. This depends on such parameters as the topology of the parallel computer and the relative values of communication set-up times and unit transmission and computation costs. Certainly, for problems in which the cost of dividing the subtasks is linear, redistribution costs are significant and Concatenated Parallelism is beneficial. However, this may be true for practical values of problem sizes even when the subdivision cost is not linear but a function close to linear. We have shown several important problems for which Concatenated Parallelism has advantages – quicksort, quickhull, construction of quadtrees, octrees and multidimensional binary search trees.

The advantage of Concatenated Parallelism over Task Parallelism (when such an advantage exists) depends upon the nature of the divide and conquer tree. For balanced tree, the redistribution cost is reduced by a factor of $\log p$ on a hypercube while there is no advantages on a mesh. For unbalanced problems, the advantage gained depends upon the effect of imbalance. In the worst case, Task Parallelism fails to provide any speedup at all while Concatenated Parallelism always provides linear speedup. Task Parallelism is sensitive to imbalance where as imbalance has no effect on Concatenated Parallelism. For randomized divide and conquer trees, Concatenated Parallelism takes advantage of the fact that the variance of the sizes of the subtasks at the i^{th} level of the divide and conquer tree reduces with increase in i. Since redistribution is performed only at the last stage, the subtask sizes allocated to individual processors are relatively uniform. Task Parallelism is affected due to high variance in subtasks sizes close to the root of the divide and conquer tree. This can be remedied by using an allocation of processors proportional to the individual subtask sizes but this strategy will not yield ideal results because allocation of processors can not be fractional and allocation that does not respect the topology often leads to congestion problems. It may also be unnatural and hard to program.

There are still several issues that remain to be investigated. There is considerable choice in redistribution strategies for Concatenated Parallelism. Redistribution should be done as soon as a balanced allocation to individual processors is possible, in order to extract the full benefits of Concatenated Parallelism. It would be interesting to investigate provably optimal redistribution strategies. Another interesting avenue to explore is a hybrid approach combining Concatenated Parallelism with Task Parallelism. One strategy is to continue with Concatenated Parallelism until the communication cost at the next stage of subdivision exceeds redistribution cost. At this stage, processors should be grouped into as many groups of equal size as possible such that a fair redistribution can be done. After the redistribution, the same strategy is recursively applied to each group. Such a strategy can potentially obtain the full benefits of both Concatenated and Task parallelism by dynamically switching between the strategies during the execution of the divide and conquer algorithm.

References

[1] I. Al-furaih, S. Aluru, S. Goil and S. Ranka, Parallel Construction of multidimensional binary search trees, *To appear in Proc. International Conference on Supercomputing*, Philadelphia. (1996)

[2] I. Al-furaih, S. Aluru, S. Goil and S. Ranka, Practical Parallel Algorithms for Selection on Coarse-Grained Parallel Computers, *To appear in Proc. International Parallel Processing Symposium*, Honolulu. (1996)

[3] K. Belkhale and P. Banerjee, An approximate algorithm for the partitionable independent task scheduling problem, *Proc. International Conference on Parallel Processing* (1990).

[4] S. Chakrabarti, J. Demmel and K. Yelick, Modeling the benefits of mixed data and task parallelism, Computer Science Division, University of California, Berkeley.

[5] S. Chatterjee, Compiling data-parallel programs for efficient execution of shared-memory multiprocessors, Technical Report No. CMU-CS-91-189, Carnegie Mellon University, Pittsburgh, PA, 1991.

[6] A. Feldmann, J. Sgall, and S.H. Teng, Dynamic scheduling on parallel machines, *Foundations of Computer Science* (1992) 111-120.

[7] G. Fox, et. al. *Solving Problems on Concurrent Processors: Volume I - General Techniques and Regular Problems.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[8] R. M. Karp, Probabilistic Recurrence Relations, *Journal of the ACM*, Vol.41, No.6, pp. 1136-1150, November 1994.

[9] V. Kumar, A. Grama, A. Gupta and G. Karypis, Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin Cummings Publishing Company, California, 1994.

[10] Z. Li and E. M. Reingold, Solution of a divide-and-conquer maximin recurrence, *SIAM Journal on Computing*, Vol.18, No.6, pp 1188-1200, December 1989.

[11] W. Ludwig and P. Tiwari, Scheduling malleable and nonmalleable parallel tasks, *Proc. Symposium on Discrete Algorithms* (1994) 167-176.

[12] F.P. Preparata and M.I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, New York, 1985.

[13] S. Ramaswamy, S. Sapatnekar and P. Banerjee, A convex programming approach for exploiting data and functional parallelism on distributed memory multiprocessors, *Proc. International Conference on Parallel Processing* (1994).

[14] S. Ranka, R.V. Shankar and K.A. Alsabti, Many-to-many communication with bounded traffic, *Proc. Frontiers of Massively Parallel Computation* (1995).

[15] H. Samet, Design and Analysis of Spatial Data Structures, Addison-Wesley Publishing Company, 1990.

[16] J. Subhlok, J. Stichnoth, D. O'Hallaron and T. Gross, Exploiting task and data parallelism on a multicomputer, *Proc. Principles and Practices of Parallel Programming* (1993) 13-22.

[17] J. Turek, J.L. Wolf, and P.S. Yu, Approximate algorithms for scheduling parallelizable tasks, *Proc. Symposium on Parallel Algorithms and Architecture* (1992) 323-332.