Cooperative Task-Oriented Computing: Algorithms & Complexity
shown that using algorithm $X$ with $p$ processors to solve the Do-All problem of size $n$ has the same asymptotic work complexity as of using algorithm $X$ with $p$ processors to solve $\frac{n}{p}$ Do-All problems of size $p$ one at a time.

We prove the complexity of solving the $\frac{n}{p}$ Do-All problems of size $p$. By Lemma 3.8, $S_{p,p} = O(p \cdot p^{\log \frac{1}{2}}) = O(p^{\log 3})$. Thus the overall work is $S = O\left(\frac{n}{p} S_{p,p}\right) = O\left(\frac{n}{p} p^{\log 3}\right) = O(n \cdot p^{\log \frac{1}{2}})$, as desired.

3.3 ALGORITHM Groote

We now present another asynchronous algorithm for the Do-All problem which we call Groote. We first present in an incremental way the idea behind the algorithm and then we describe the complete algorithm.

3.3.1 A High-level View of the Algorithm.

The algorithm is a generalization of the two-processor algorithm based on processor collisions (recall Example 2.2): a situation where two (or more processors) concurrently and redundantly perform the same task and then detect that they have “collided” on this task. The tasks are represented in a linear array and are numbered from 1 to $n$. The first processor inspects and then performs tasks sequentially starting at task number 1 and moving up; the second processor inspects and then performs tasks sequentially starting at task number $n$ and moving down. Each processor stops when it inspects a task that has already been performed by the other processor. As we show in the next subsection, the work spent for inspecting a task for completion is negligible compared to the work spent for performing a task. We denote by $C$ the cost spent for performing the tasks and $C_2$ the cost by two processors. Then, it is not difficult to see that $C_2$ is at most $n + 1$: each task but the final one is performed exactly once, while the final task is performed at most twice if both processors find it not done concurrently.

We now consider the Do-All problem for 4 processors and $n$ tasks. We assume that $n$ is perfect square, i.e., $n = m^2$ for some $m$ (if $n$ is not a perfect square the usual padding approach can be taken). We arrange the tasks in a 2-dimensional $m$ by $m$ array. The processors work in pairs on each row of the matrix: the first processor pair starts at the top row and moves down, the second pair starts at the bottom row and moves up. When working on a row, each pair of processors follows the two-processor algorithm described above. Prior to starting on a new row, each pair checks whether or not the next row is done, if so, the algorithm terminates: either all rows are now done, or the two pairs have collided on a row also meaning that all rows are done. Figure 3.5 illustrates the algorithm.

Using the analysis for the two-processor algorithm, the cost of the work of each pair on a single row of tasks is no more than $m + 1$. There are $m$ rows and when the
4.3. Algorithm Groote

Four-processor algorithm for \( n = 64 = 8^2 \). Each pair works on one row at a time (i.e., a one-dimensional instance of the problem). The algorithm terminates when the two pairs of processors “meet” (collide).

Figure 3.5: Algorithm for 4 processors.

Eight-processor algorithm for \( n = 512 = 8^3 \). Each quadruple works on one 8 by 8 slice of the cube (i.e., a two-dimensional instance of the problem) at a time. The algorithm terminates when the two quadruples “meet” (collide).

Figure 3.6: Algorithm for 8 processors.

two pairs of processors collide the final row is done twice. Thus in the worst case the number of rows of tasks performed by the 4 processors is \( m + 1 \). The total cost \( C_4 \) of the work is therefore: \( C_4 = C_2 \cdot (m + 1) = (m + 1) \cdot (m + 1) = (m + 1)^2 \).

This idea can be extended to 8 processors as follows. We assume that \( n \) is perfect cube, i.e., \( n = m^3 \) for some \( m \) (if \( n \) is not a perfect cube the usual padding approach can be taken). We arrange the tasks in a 3-dimensional \( m \) by \( m \) by \( m \) array. The processors work in quadruples on each 2-dimensional “slice” of the cube: the first processor quadruple starts at the front surface of the cube and moves “in”, the second quadruple starts at the read surface and moves “out”, one slice at a time. When working on a slice, each quadruple of processors follows the 4-processor algorithm described above. Prior to starting on a new slice, each quadruple (i.e., each processor in the quadruple) checks whether or not the next slice is done, if so, the algorithm terminates: either all 2-dimensional slices of the cube are now done, or the two quadruples have collided on a slice also meaning that all tasks are done. Figure 3.6 illustrates the algorithm.
Using the analysis for the 4-processor algorithm, the cost of the work of each quadruple on a single slice of tasks is no more than $C_4 = (m + 1)^2$. There are $m$ 2-dimensional slices and when the two quadruples of processors collide the final slice is done twice (at most). Thus in the worst case the number of slices of tasks performed by the 8 processors is $m + 1$. The total cost $C_8$ of the work is therefore: $C_8 = C_4 \cdot (m + 1) = (m + 1)^2 \cdot (m + 1) = (m + 1)^3$.

As one can see, this algorithm implements the local allocation paradigm using a multi-level approach similar to hierarchical trees as described in Section 2.2.1 (paragraph “Techniques for Improving Performance” – Hierarchical Tree Traversals). We next present the algorithm for many processors.

### 3.3.2 The Algorithm for $p = 2^k$ and $n = m^k$

This idea is generalized to $p = 2^k$ processors as follows. We assume that $n = m^k$ for some parameter $m$. We arrange the tasks in a $k$-dimensional array.

The processors work in groups of $p/2 = 2^{k-1}$ processors on each 2-dimensional “slice” of the $k$-cube: the first group starts at the “top surface” of the cube and moves “down”, the second quadruple starts at the “bottom surface” and moves “up”, one slice at a time. Note that each slice is a $(k - 1)$-dimensional array. When working on a $(k - 1)$-dimensional slice, each group of processors performs the $(2k - 2)$-processor algorithm. Prior to starting on a new slice, each group (i.e., each processor in the group) checks whether or not the next slice is done, if so, the algorithm terminates: either all slices of the $k$-cube are now done, or the two groups have collided on a slice also meaning that all tasks are done. The detailed pseudocode for the algorithm is left as an exercise (see Exercise 3.8).

We now analyze the complexity of algorithm Groote. We first prove the cost for performing tasks and then the cost for checking for completion of slices.

**Lemma 3.10** Algorithm Groote solves the Do-All problem with $n = m^k$ tasks and $p = 2^k$ asynchronous processors using task-based cost $C = O(np^c)$, where $0 < c = \log((m+1)/m) < 1$.

**Proof.** We first compute $C_p$ by iteratively computing the cost on the $k$ dimensions. In particular, the total work $C_p$ is computed as follows:

$$C_p = C_{p/2} \cdot (m + 1) = C_{p/4} \cdot (m + 1) \cdot (m + 1) = \ldots = C_2 \cdot (m + 1)^{k-1} = (m + 1)^k.$$  

Now, to express the cost $C$ in terms of $p = 2^k$ and $n = m^k$ we perform the following reformulation:

$$C = (m + 1)^k = n \cdot \left(\frac{m + 1}{m}\right)^k = n \cdot \left(\frac{m + 1}{m}\right)^{\log p} = n \cdot p^{\log \left(\frac{m+1}{m}\right)}.$$  

Hence, when $m$ is a constant parameter (as one would expect), we have $0 < \log((m + 1)/m) < 1$, and so the algorithm has subquadratic cost, as desired.

\[\square\]
Checking for completion of slices involves in reading and writing bits. We let \( B \) denote the bit-based cost spent for checking for completion of slices.

Lemma 3.11 Algorithm Groote solves the Do-All problem with \( n = m^k \) tasks and \( p = 2^k \) asynchronous processors using bit-based cost \( B = O(n \log p) \).

**Proof.** We need to compute the number of bits that need to be examined. From the description of the algorithm it follows that with dimension \( k \), there are \( m^k + m^{k-1} + \ldots + m^1 \) bits to be examined. However, not all of them are examined by the same number of processors.

As mentioned in the description of the algorithm, the processors are divided into groups, and when working on a \((k-1)\)-dimensional slice, each group of processors performs the \((2k - 2)\)-processor algorithm. Given a one dimensional array, it can be seen that in the worst case \( m + 2 \) bits must be examined: each processor checks \( m/2 \) bits and two extra bits are checked for the processors to realize that the whole array has been checked.

Then, at the highest level there is 1 (\( = m^0 \)) instance of \( m + 2 \) bits examined by \( 2^{k-1} \) processors each. At the next level there are \( m \) instances of \( m + 2 \) bits examined by \( 2^{k-2} \) processors each. At the next level there are \( m^2 \) instances of \( m + 2 \) bits examined by \( 2^{k-3} \) processors each and so on. Hence, at the lowest level there are \( m^{k-1} \) instances of \( m + 2 \) bits examined by 1 (\( = 2^0 \)) processors each. Summing these gives us the total number of bits examined. Note that each bit examination might incur two work units (one for reading and one for writing) in the worst case. Therefore,

\[
B \leq 2 \sum_{i=0}^{k-1} m^{k-i}(m + 2)2^{i-1} = 2(m + 2) \sum_{i=0}^{k-1} m^{k-i}2^{i-1}. \tag{3.1}
\]

We consider two cases.

*Case 1: \( m = 2 \) (i.e., \( p = n \)).* Replacing \( m \) with 2 in Equation (3.1), we get that

\[
B \leq 8k2^{k-1} = 4k2^k = 4n \log p.
\]

*Case 2: \( m > 2 \).* Replacing \( 2^{i-1} \) with \( m^{i-1} \) in Equation (3.1) and since \( m + 2 \leq 2m \) we get that

\[
B \leq 4mkm^{k-1} = 4km^k = 4n \log p.
\]

Putting the two cases together we get the thesis of the lemma.

The work \( S \) of the algorithm is the sum of \( C \) and \( B \). From Lemmas 3.10 and 3.11 it can be seen that \( B \) is asymptotically subsumed by \( C \). This gives the main complexity result.
Theorem 3.12 Algorithm Groote solves the Do-All problem with \( n = m^k \) tasks and \( p = 2^k \) asynchronous processors \((p \leq n)\) using work \( S = O(np^c)\), where \( 0 < c = \log\left(\frac{m+1}{m}\right) < 1 \).

Note that if we want \( n = p \), we choose \( m = 2 \). In this case the work becomes \( O(n^{\log 3})\). Earlier we studied a binary tree based algorithm, called algorithm X. As we saw, its work for \( n = p \) is also \( O(n^{\log 3})\). This is not surprising – algorithm X is very similar in its approach to the algorithm we described here – there the binary progress tree is used to control the execution, where a “collision” occurs when some two subtrees are completed.

### 3.4 ALGORITHM AW\(^T\)

In this section we present an algorithm for the asynchronous Do-All problem that uses generalized progress trees for detection of unfinished work in the style of the local allocation paradigm, and that uses permutation schedules to select potential work in the style of the hashed allocation paradigm. We call this algorithm, algorithm AW\(^T\).

As we show, the data structures of algorithm AW\(^T\) can be parameterized, so that its work is \( S = O(n^{1+\epsilon}) \) for \( p = n \), and for any \( \epsilon > 0 \). Hence, this algorithm has better work complexity than the other asynchronous algorithms we have seen so far.

Recall that in algorithm X, the processors traverse the work elements in the order determined by the processors’ PIDs and the structure of the progress tree (which is a binary tree). At each internal node of the tree, a processor chooses between the two children based on whether any of the children are not “done”, and based on the bit in that processor’s PID corresponding to the depth of the node. This bit can be interpreted as a permutation of the two children. When a processor arrives at a node, it uses one of the two permutations to choose the order of the progress tree traversal. The choice of the permutation is based on the bit in the binary expansion of the processor’s PID in position corresponding to the depth of the node.

Algorithm AW\(^T\) generalizes this approach using a \( q \)-ary progress tree \((q \geq 2)\) and permutations for the processors to traverse the tree. Before we describe algorithm AW\(^T\) we define the term contention of permutation schedules and explain how this is related with redundant work.

#### 3.4.1 Contention of Permutations

In this section we present and generalize the notion of contention of permutations and give properties of contention (without proofs). Contention properties turn out to be important in the analysis of Algorithm AW\(^T\) as well as other algorithms presented in later sections.
Bibliography


120 BIBLIOGRAPHY


