Multicore Parallel Min-Cost Flow Algorithm for CAD Applications

Yinghai Lu[1], Hai Zhou[2,1], Li Shang[3], Xuan Zeng[1*]
[1]State Key Lab of ASIC & System, Microelectronics Dept., Fudan University, China
[2]EECS, Northwestern University, U.S.A., [3]ECEE, University of Colorado, Boulder, U.S.A.

Abstract—**Computational complexity has been the primary challenge of many VLSI CAD applications. The emerging multicore and many-core microprocessors have the potential to offer scalable performance improvement. How to explore the multicore resources to speed up CAD applications is thus a natural question but also a huge challenge for CAD researchers. Indeed, decades of work on general-purpose compilation approaches that automatically extracts parallelism from a sequential program has shown limited success. Past work has shown that programming model and algorithm design methods have a great influence on usable parallelism. In this paper, we propose a methodology to explore concurrency via nondeterministic transactional algorithm design, and to program them on multicore processors for CAD applications. We apply the proposed methodology to the min-cost flow problem which has been identified as the key problem in many design optimizations, from wire-length optimization in detailed placement to timing-constrained voltage assignment. A concurrent algorithm and its implementation on multicore processors for min-cost flow have been developed based on the methodology. Experiments on voltage island generation in floorplanning demonstrated its efficiency and scalable speedup over different number of cores.**

Categories and Subject Descriptors:
J.6 [Computer-Aided Engineering]: Computer-Aided Design
D.1.3 [Programming Techniques]: Concurrent Programming – Parallel programming
General Terms: Algorithms, Performance
Keywords: Min-cost flow, Multicore, Parallel programming

I. Introduction

VLSI computer-aided design (CAD) software for multi-billion transistor IC design has become increasingly complex and requires more and more computation resources. This challenge can potentially be mitigated by emerging multicore and manycore systems. Since 2004, multicore microprocessor has become the main engine of mainstream servers and personal computers [6], [7]. Nowadays, it is rare to see uni-core processors even in laptop computers, and servers often come with eight cores on one or two CPUs. Therefore, it is natural to hope that manycores available in modern computers may be effectively utilized to speed up CAD programs. However, numerous unsuccessful past attempts have shown that, programming model has great influence on usable parallelism; without exploring concurrency in algorithm design, it is impossible to achieve reasonable speedup in multicore or manycore systems.

Recently, multicore parallel CAD has drawn significant attention in the design automation field [2], [5], [21], [30]. Various existing techniques to explore program concurrency have been borrowed for parallel CAD programming.

Automated parallelization is a compilation approach that extracts parallelism from a sequential program. It has been extensively investigated for many years, but has shown limited success [26]. The general consensus in the community is that a program's automatically-exploitable concurrency is generally fixed by the programming or the programmer's way of thinking. Conventional sequential programming heavily limits a program's usable concurrency.

Message passing approaches such as MPI [24] explicitly implement a computation by multiple processes that work in separate memory spaces and synchronize via passing messages. It is easy to understand. However, the programming model is at a low abstraction level, closer to the physical platform. Similar to assembly language programming, it requires the programmer to think in physical details and the (even more difficult) concurrent execution of processes. Furthermore, such a program needs to be redesigned for different generations of many-core processors.

Threading (or multithreading) implements a computation using multiple threads that share a common memory space and can be executed concurrently. Thread synchronizations are most commonly achieved by locking. However, coarse-grain locking does not perform well, while fine-grain locking is error-prone. Common problems in fine-grain locking include deadlock and the inability to compose program fragments that are correct in isolation [9]. In addition, it is not known how a programmer can come up with a multithreaded program with correctness guarantee.

Transactional memory is a shared memory model proposed by Herlihy et al. [10]. Instead of locking a set of memory elements before accessing them, a program using transactional memory marks some blocks of instructions *transactions*. A transaction is a state change that happens *atomically*. A runtime implementation of transactional memory, either via software or hardware, may speculatively execute many transactions in different threads. If there is no conflict among these transactions, concurrency is effectively explored. Otherwise, only one of the conflicting transactions is permitted to take effect and the others are aborted.

In this paper, we identify the nondeterministic transactional programming model (as in UNITY [3] and TLA+ [16]) as the most effective algorithm design approach for exploring concurrency. There is a systematic algorithm design methodology for such a model that is natural for problem solving and constructs an algorithm together with its correctness proof. More importantly, the correctness of the algorithm allows nondeterminacy in execution order of the commands. Two such commands can be executed concurrently if there is no conflict. We believe that the nondeterministic atomic command model is perhaps the closest to concurrency while still manageable in our brain. We also advocate a design principle to create many small local commands for concurrency exploration. Philosophically, program parallelization is a trade-off between communication and load balancing. Optimizing both is often possible, if we create many short actions with nondeterministic order. Starting with such an algorithm, we also develop an approach to programming it in such a way that we only need to program and compile it once to run on machines with different cores. We believe such a feature is important to achieve scalable performance speedup on multicore architectures which will double the core number in each generation.

To test and demonstrate such a multicore CAD programming method, we select the min-cost network flow problem and develop a multicore parallel program for it. There are tens of different CAD

[*]Corresponding author. E-mail: xzeng@fudan.edu.cn

DAC 2009, July 26 - 31, 2009, San Francisco, California, USA
Copyright 2009 ACM ACM 978-1-60558-497-3 -6/08/0006 ...$5.00.

problems that can be formulated as min-cost flow problem or solved via min-cost flow as the key subroutine. These include voltage assignment [20], gate sizing [28], clock skew optimization [18], retiming [29], floorplan area minimization [19], placement wire minimization [27], etc. In general, many CAD problems need to minimize a weighted summation of element costs under the timing constraint in terms of the longest path delay or the maximal delay-to-register ratio on the cycles. Such a problem is very close in structure to the dual problem of the min-cost flow, as will be explained in Section II. There was theoretical study of parallel algorithms for min-cost flow problem [22]. However, practical multicore parallel program is still needed.

One of the most recent applications of the min-cost flow technique to CAD is the voltage assignment in voltage island floorplanning [20], [17], [12]. Different supply voltages need to be assigned to different blocks to minimize the total power consumption under the constraint that the longest delay is upper bounded. Such a problem can be formulated as the dual problem of the convex-cost network flow, which is then translated into a min-cost flow problem [20]. We apply our multicore parallel program to the voltage island floorplanning problem and demonstrate the effectiveness and scalability of our approach.

The contributions of the paper include a nondeterministic transactional algorithm design method for exploring concurrency, a systematic approach to program such an algorithm for multicore platforms, their application in developing an efficient multicore program for the min-cost flow problem, and the application of the program to speed up voltage island floorplanning. Furthermore, such an approach is applicable to other algorithmic problems in CAD.

The rest of the paper is organized as follows. In Section II, the voltage island assignment problem is formulated as the dual of min-cost flow problem. A nondeterministic transactional algorithm for min-cost flow problem is developed in Section III, and the general methodology for mapping such a transactional algorithm to parallel program on a multicore platform is described in IV. Speedup improvement techniques for solving the voltage assignment problem on multicore is introduced in V. The effectiveness of the proposed methodology is demonstrated through experiments in Section VI. Finally, the paper is concluded in Section VII.

II. Problem Formulation

Many CAD problems, especially timing-constrained design optimization problems, center around minimizing implementation cost under various timing constraints. When signal arrival time in the design is considered, and the implementation cost of an element is a function of its delay, such a problem can usually be formulated as the following mathematical program.

$$\begin{aligned} Min \quad & \sum_{(i,j)\in E} cost_{ij}(d(i,j)) \\ s.t. \quad & \forall (i,j)\in E : p(i) + d(i,j) \le p(j) \qquad (1) \end{aligned}$$

where the decision variables are p (arrival time) and d (element delay). The function $cost_{ij}(d)$ computes the element cost to achieve a delay of d. It is a non-increasing function, and is usually convex, meaning that speeding up an element is more difficult at higher speed zone.

Recently, voltage island generation has become an important CAD problem because of the thermal issues in modern VLSI designs [20]. Multiple different supply voltages will be applied to different blocks to minimize the power consumption while satisfying the timing constraints. It is also desirable that blocks with the same supply voltages be placed together in order to minimize the power/ground network. The central problem in voltage island generation is the **timing-constrained voltage assignment problem** which can be formulated as follows.

$$\begin{aligned} Min \quad & \sum_{(i,j)\in E} power_{ij}(v(i,j)) \\ s.t. \quad & \forall (i,j)\in E : p(i) + d_{ij}(v(i,j)) \le p(j) \qquad (2) \\ & \forall i \in V : 0 \le p(i) \le \phi \qquad (3) \\ & \forall (i,j)\in E : v(i,j) \in Voltage \qquad (4) \end{aligned}$$

where v is the supply voltage, and both the delay d and the power consumption $power$ are its functions. When v is treated as (inverse) function of d, $power$ can be represented as a function of d; the above formulation becomes very close to our general formulation. The only exceptions are the bound constraints on the arrival time p and the discrete voltage requirements in the last two formulas. The bound constraints can be subsumed into the difference inequalities with an introduced ground node. The idea is to treat the lower bound as $p(O) + 0 \le p(i)$ and the upper bound as $p(i) - \phi \le p(O)$, where $p(O)$ for the ground node O is always 0. The discrete requirement will be first relaxed to get a continuous solution, which will then be rounded by a heuristic.

The continuous timing-constrained voltage assignment problem and the general formulation as discussed above can be translated into the following formulation, which is the dual of the min-cost network flow problem.

$$\begin{aligned} Max \quad & \sum_{(i,j)\in E} c(i,j)d(i,j) \\ s.t. \quad & \forall (i,j)\in E : p(i) + d(i,j) - w(i,j) \le p(j) \qquad (5) \end{aligned}$$

Its dual, the min-cost flow problem, is given as follows.

$$\begin{aligned} Min \quad & \sum_{(i,j)\in E} w(i,j)f(i,j) \\ s.t. \quad & \forall (i,j)\in E : 0 \le f(i,j) \le c(i,j) \qquad (6) \\ & \forall j \in V : \sum_{(i,j)\in E} f(i,j) = \sum_{(j,k)\in E} f(j,k) \qquad (7) \end{aligned}$$

The Karush-Kuhn-Tucker condition [25] for both the primal and dual problems is the same and given as follows.

$$\begin{aligned} P0 \quad & \triangleq \quad \forall (i,j)\in E : 0 \le f(i,j) \le c(i,j) \qquad (8) \\ P1 \quad & \triangleq \quad \forall j \in V : \sum_{(i,j)\in E} f(i,j) = \sum_{(j,k)\in E} f(j,k) \qquad (9) \\ P2 \quad & \triangleq \quad \forall (i,j)\in E : (f(i,j) < c(i,j) \Rightarrow p(i) - w(i,j) \le p(j)) \\ & \qquad \wedge (f(i,j) > 0 \Rightarrow p(i) - w(i,j) \ge p(j)) \qquad (10) \end{aligned}$$

They are necessary and sufficient conditions for both the primal and the dual problems. Any correct algorithm need to satisfy them as its post-conditions, which must be true at the end of the algorithm.

III. Nondeterministic Transactional Algorithm Design for Concurrency

We propose to use a **nondeterministic transactional programming** method to explore concurrency in algorithm design. Such a method can be traced back to Dijkstra's guarded commands [4], which had been later developed into UNITY [3]. In UNITY, every algorithm is composed of an initialization followed by a loop of guarded commands. No order is imposed on the commands. When the condition (i.e., guard) is valid for a command, it can be selected for execution. The correctness of the algorithm does not depend on the execution order of the commands, but depends on the atomic execution of each command. Lamport further demonstrated in his TLA [16] that such a model can be used to naturally specify any system including reactive systems.

We believe nondeterministic transactional programming is suitable for multicore algorithm design, based on the following reasons.

First, thinking and reasoning about arbitrary asynchronous concurrent actions are difficult for human brains, since the possibilities are exponential. Fortunately, reasoning on isolated actions without restricting their ordering is within our capabilities, thanks to the concept of invariant and mathematical induction. Second, there is a systematic algorithm design method for this model that employs assertional proof techniques [23], [15] to guarantee correctness, as the fruit of many years' research on programming theory. Finally, high performance can be achieved by concurrently executing many actions, as long as conflicts are rare, which is common in practice and has been confirmed by transactional memory research [10]. Therefore, *one principle we use to explore concurrency in algorithm design is to produce as many small actions as possible.*

We will now develop an algorithm for the min-cost flow problem using the nondeterministic transactional programming method to explore concurrency. The post-condition of the algorithm is $P0 \wedge P1 \wedge P2$ as defined in the previous section. An important design decision in the method is to select a predicate within the post-condition as the invariant and use the remaining as the loop goal. Here we simply select $P0$ as the invariant which can be easily satisfied by an initialization $f := 0$.

When $P1$ is not true, there must be at least two nodes whose inflows are not equal to their out-flows. The node excess is defined as

$$X(j) \triangleq \sum_{(i,j)\in E} f(i,j) - \sum_{(j,k)\in E} f(j,k).$$

A node with positive excess is defined to be active. Under a given f, the residual edges $E(f)$ are defined as the edges where an extra flow can be added. Decreasing a flow on an edge is equal to adding the flow on the reverse direction. Formally,

$$\begin{aligned} E(f) \quad &\triangleq \quad \{(i,j) | (i,j) \in E \wedge f(i,j) < c(i,j)\} \\ &\quad\quad \cup \{(j,i) | (i,j) \in E \wedge f(i,j) > 0\}. \end{aligned}$$

The residual capacity $c_f(i,j)$ of an residual edge (i,j) is defined as $c(i,j) - f(i,j)$ if it is in E (i.e., forward edge) or $f(j,i)$ if it is not (i.e., backward edge). For an active node i, we want to push flows from i over residual edges to its neighbors. But such an operation may introduce a residual edge in the reverse direction. We define the reduced cost of an edge (i,j) as

$$w^p(i,j) \triangleq w(i,j) - p(i) + p(j).$$

With the definitions, the post-condition $P2$ can be simplified as

$$P2 = \forall (i,j) \in E(f) : w^p(i,j) \geq 0.$$

To satisfy $P1$ while not to violate $P2$, we only push flow over (i,j) with $w^p(i,j) < 0$, which is called an admissible edge, giving the first guarded command in the algorithm (Figure 1). When there is no admissible edge from an active i, we will relabel it by increasing $p(i)$ by $\epsilon/2$, giving the second command. When $P2$ is not true, there will be a residual edge (i,j) with $w^P(i,j) < -\epsilon$. We can simply remove such an edge by filling its capacity, giving the third command.

Now a tricky problem is how to decide the potential changing step ϵ in the second command. A larger step will render more valid residual edges for pushing flows out of i, but may give more residual edges violating $P2$. As a trade-off, we can gradually reduce ϵ until $\epsilon < 1/|V|$, giving the fourth command. In summary, the whole algorithm is given in Figure 1, where

$$P2(\epsilon) \triangleq \forall (i,j) \in E(f) : p(i) - w(i,j) \leq p(j) + \epsilon.$$

It is actually Goldberg's min-cost flow algorithm, and its correctness and complexity are given in the following theorem. Such an algorithm exposes much concurrency since it has $2|E| + |V| + 1$ actions.

Theorem 1: Goldberg's algorithm in Figure 1 is correct under nondeterministic atomic execution of the guarded commands. The number of iterations is upper bounded by $O(|V|^2|E|log(|V| \max_{(i,j)\in E} |w(i,j)|))$.

```
$f, p, \epsilon := 0, 0, \max_{(i,j)\in E} |w(i,j)|$
do $\{P0\}$
    $\exists (i,j) \in E(f) : X(i) > 0 \wedge -\epsilon \leq w^p(i,j) < 0$
        $\rightarrow push(i,j)$
    $\exists i \in V : X(i) > 0 \wedge \forall (i,j) \in E(f) : w^p(i,j) \geq 0$
        $\rightarrow p(i) := p(i) + \epsilon/2$
    $\exists (i,j) \in E(f) : w^p(i,j) < -\epsilon$
        $\rightarrow f(i,j) := f(i,j) + c_f(i,j)$
    $P1 \wedge P2(\epsilon) \wedge \epsilon \geq 1/|V| \rightarrow \epsilon := \epsilon/2$
od $\{P0 \wedge P1 \wedge P2(\epsilon) \wedge \epsilon < 1/|V|\}$
```

Fig. 1. Nondeterministic transactional algorithm for min-cost flow.

IV. Multicore Programming of Nondeterministic Transactional Algorithm

Even though a nondeterministic transactional algorithm can be naturally designed and formally proved correct, as demonstrated in the previous section, no current existing programming platform supports it. In this section, we will leverage existing mechanism and develop a systematic approach for mapping such an algorithm to a multithreading program for multicore platforms. An outstanding feature of the approach is that the program only needs to be developed once and can then take advantage of different number of cores in a platform.

A. General Methodology

It is tempting to create a thread for each guarded command in the transactional algorithm; the many number of threads will be automatically scheduled by the OS to available cores, and the atomicity can be enforced with simple locking or transactional memory. However, since the OS has no knowledge of the operations in the threads, it might preempt a thread doing useful thing by a busy-waiting one. Even worse, a thread could be preempted in the middle of a (atomic) transaction, increasing the chance of conflict or abortion, not to mention the overhead of scheduling and managing the threads. Another plausible approach is to have one thread take care of a group of guarded commands. For example, four threads may be created for the four groups of commands in Goldberg's algorithm in Figure 1. However, due to instance-dependent dynamics of execution, the number of iterations for each command is unknown and could change dramatically, making load balance hard to achieve, if not impossible. The same is true for static task partitioning based on data (for example, on V or E in Figure 1).

After careful examination of many plausible ideas, we propose the following general method to implement a nondeterministic transactional algorithm on a multicore platform: Each core will have all the code and be able to execute every guarded command; which command is executed will depend on which data are available and controlled by each core. The data (or their tokens) will be moved dynamically among the cores. Because of the universality of the cores, no data need to be moved if not for the purpose of load balancing. Data affinity is thus preserved.

We leverage the mechanism of multithreading to create a thread for each core. Such a thread is created at the beginning of the program and will last till the end of the program. It is ideal for each thread to keep running on a core during the program execution. From now on, core and thread will be used interchangeably. In order to manage the control of data (or tasks) among the threads, a queue will be maintained as a globally shared data structure. Each thread will fetch control tokens from the queue when it finishes its current work, and will release tokens to the queue based on its schedule. Dynamic load balancing is thus attainable through self-discipline. Since the execution of a command will change the state thus may render more valid guards, all threads need to be synchronized in order to detect that all guards are false thus the program can be terminated.

Applying the general method, we can map Goldberg's algorithm in Figure 1 into a multithreaded program with the identical thread program in Figure 2. We maintain a global queue Q to hold the nodes whose excess flow or reduced cost condition will enable the corresponding push/relabel commands. A thread repeatedly tries to fetch a bunch of nodes into its local input buffer q_{in}. Then, it exams the nodes in q_{in} and their associated edges one by one to see whether they enable the first three groups of commands in Figure 1. If a guarded command is enabled, the thread carries out the corresponding action such as push, relabel or fill on the node or edge. The actions, as are parenthesized in Figure 2, should be executed as an atomic transaction, in order to guarantee the correctness of the program. The atomicity can be achieved by using conventional mutual exclusion or modern transactional memory [10]. An action such as push will probably introduce more valid guarded commands because pushing a flow toward a node may make it active. The newly active nodes are stored in the local output buffer q_{out} and are later on flushed to Q.

If a thread fails to fetch any active nodes from Q, which means that Q is currently empty, it becomes idle. Then it tries to synchronize with other threads in order to know their status. The detail of global synchronization (*Sync on idle* in Figure 2) will be discussed in the next subsection. When all the threads are idle, the fourth guarded command is enabled, and ϵ is reduced. With reduced ϵ, the first three group of guarded commands may becomes valid again. To enable their examination, all nodes will be activated and added to Q. Each thread will repeat the process until $\epsilon < 1/|V|$. Then all threads terminate and the program completes.

```
while $\epsilon > 1/|V|$
    if get some active nodes $V_a$
        for $i \in V_a$
            for $(i,j) \in E(f)$
                {if $(w^p(i,j) < -\epsilon)$ $f(i,j) := f(i,j) + c_f(i,j)$
                elseif $(w^p(i,j) < 0)$ $push(i,j)$}
            end for
            if $(X(i) > 0)$ {$relabel(i)$}
        end for
    elseif Sync on idle
        $\epsilon := \epsilon/2$
        activate $V$
end while
```

Fig. 2. The program for each core/thread.

Our implementation of transactional algorithm on multicore machine overcomes many disadvantages of other possible approaches discussed above. Firstly, with long-living threads, the overhead of thread creation and termination is largely reduced. Second, since each thread is bound with a core, it is much less likely to be preempted during execution, reducing overhead and half-complete transactions. Furthermore, instead of being idle, each thread (core) is constantly making progress or checking for new valid guarded commands. All threads terminate almost simultaneously when there are no valid guarded commands. The flexibility of task-based programming facilitates load balance tuning of our program using the technique introduced later.

B. Global Synchronization

According to previous discussions, the fourth command in Figure 1 become valid only when all the other three guarded commands are not enabled. However, a thread cannot check the validation of the fourth command by simply checking the status of global queue to see whether it is empty, because some other threads may still be processing and their operations may re-enable some guarded commands. In the context of Goldberg's algorithm, a push operation will add flow to the target node and may make it active. So besides the status of global queue, we need to introduce a global synchronization mechanism that tells each thread the status of all other threads. In out implementation, a modified version of termination detection barrier (TDBarrier) in [11] is used to do the job.

A TDBarrier contains a counter, implemented by an atomic integer register, and listens to the status of each thread. Before all the threads are launched, a TDBarrier is created and the counter is initialized to be zero. Each time a thread sets itself as idle, it decreases the counter by one; and each time it sets itself as active, it increases the counter by one. When a thread asks TDBarrier about the global thread status, the TDBarrier returns *true* if the counter equals to zero, indicating that all the threads are idle. Otherwise, the TDBarrier returns *false*, informing the querying thread that there are still other active threads and it should keep on checking the global queue for potential newly-added active nodes.

Note that the TDBarrier only does half of the job since each thread has to register its status to the TDBarrier at a proper phase of its own computation. To achieve correct synchronization, each thread registers itself as active before fetching the active nodes from Q, and as idle if the fetching fails.

Theorem 2 ([11]): The global synchronization mechanism with termination detection barrier described above guarantees correct synchronization of the threads in Figure 2.

C. Load Balancing

We have introduced a local input buffer q_{in} and a local output buffer q_{out} to hold active nodes (valid commands) in order to reduce the access to global Q. Moreover, the local buffers can be dynamically adjusted to balance the workload among different threads [1]. Due to the heterogeneity of the flow network, it is not unusual that other threads have exhausted all the active nodes in Q and are waiting for one busy thread to flush its output buffer to the global queue so that they can fetch again. In this case, it makes sense for the busy thread to shrink its q_{out} and flushes new active nodes to global queue. Conversely, if other threads are all busy and there are plenty of active nodes in Q, the size of q_{out} should grow back in order to reduce the frequency of accessing Q. The local input buffers are adjusted accordingly. Let b_k be the original buffer size for thread k, L be the size of global queue, n_{active} and n_{total} be the number of active and total threads, the dynamic load balance adjustment works as follows:

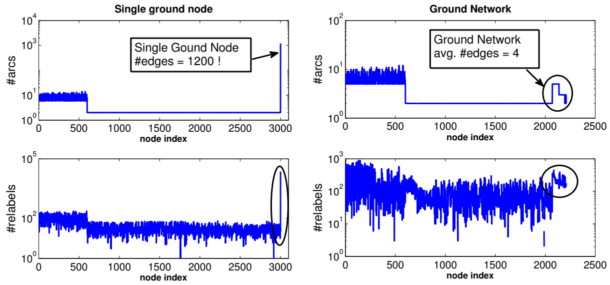
$$\begin{aligned}
&\text{if } n_{active} \leq n_{total} \times 0.75 \\
&\quad b_k = b_k/2 \\
&\text{else if } n_{active} + L/b_k \geq n_{total} \\
&\quad b_k = b_k \times 2
\end{aligned}$$

The above adjustment is carried out by each thread after processing every 100 valid guarded commands.

V. Performance Improvements

We have implemented the multicore min-cost flow solver using the techniques described in the last section, and tested it on the general graph benches [14]. The speedup in terms of the number of cores is satisfactory. However, when our algorithm is applied to solving the voltage island assignment problem, the speedup is not so good, especially in the 4-core (4C) case. Detailed examination shows that this speedup problem is due to the existence of the ground node, which is introduced to enforce bound constraints (Inequality 3) on the nodes. Figure 3(a) shows the extreme imbalance between the ground node and other nodes in terms of their connecting edges and the number of operations executed on them. Because of the high connectivity and large operation numbers of the ground node, it is highly possible that multiple cores compete for the the ground node at the same time to execute their own commands, which causes heavy contention and slows down our multicore program. Increasing the number of working cores makes the contention problem even worse.

Knowing the cause, we propose a two-fold ground network solution to address the heavy contention problem of our multicore min-cost flow solver on the voltage island assignment problem. First, as described in Section II, the function of the ground node is to



(a) Single ground (b) Ground network

Fig. 3. Performance profiling of case n300.

assert a bound on the arrival time of each block. However, it is not necessary to add such constraint to every node since a bound on the primary inputs/outputs plus the constraint in Inequality 2 guarantees that every internal block satisfies the timing constraint. So we can prune redundant edges between the ground node and the internal nodes, which reduces the number of edges and operations on the ground node.

To further balance the constraint graph, we apply a node splitting technique. The idea is to apportion the connections of IO nodes to the original single ground node to a group of ground nodes. The single ground node is first split into several ground nodes, with the connections of primary IO nodes evenly assigned to them. Then a new ground node is used to bridge all the split nodes to form a ground network with infinite capacity and zero cost. This process is demonstrated in Figure 4. Every ground node in the constructed ground network has only a few edges. When the number of primary nodes is large, we extend this idea and build a tree-structured ground network.

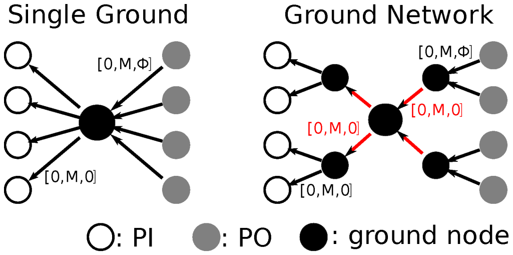


Fig. 4. Node splitting. Edge parameter: [lower cap., upper cap., cost].

After applying redundant edge pruning and node splitting, the structure and operation statistics on the constraint graph is shown in Figure 3(b). The reduction in the number of nodes is due to redundant edge pruning. When converting the convex cost flow problem into the min-cost problem, we introduce one edge for each piecewise linear part of the cost function and attach an auxiliary node with it. When the number of edges is reduced, so is the number of nodes. The circled part in Figure 3(b) corresponds to the ground network. As we can see, the number of edges and relabel operations on these ground nodes are well balanced with the other nodes in the graph. Using the proposed techniques, the heavy contention problem is solved with the balanced constraint flow graph and the multicore min-cost flow solver regains its performance on the voltage assignment problems as shown in Section VI.

VI. Experiment Results

We have implemented the multicore min-cost flow solver in C++ programming language with Intel Threading Building Blocks [13]. All the experiments are carried out on a Linux server with two dual-core 3.0GHz CPUs and 2GB RAM, which supports up to 4-core parallelism. The multicore program is compiled once and runs with a user-specified number of cores.

First, we demonstrate the effectiveness of performance improvement techniques described in Section V. Using the 4-core (4C) min-cost flow solver, we compare the average speedup and contention of the program in solving voltage assignment problems with single ground node and the proposed ground network technique. The test cases are GSRC benchmarks with additional delay-power information provide by the authors of [20] and there are four legal working voltages for each block. Because of nondeterminacy in runtime, the program is run for 10 times on every test case and the results are reported in Table I. The speedup rate is computed against the single-core program. The number of contentions is defined as the average number of data conflicts seen in a core when it commits guarded commands. Table I shows, especially for the large benchmarks, significant reduction in contention and increased speedup of the proposed multicore solver when the single ground node is replaced by the ground network.

TABLE I
Effectiveness of the Ground Network

Cases	Single Ground		Ground Network	
	#Contentions	4C Speedup	#Contentions	4C Speedup
n10	0.00	1.25	0.00	0.93
n30	58.50	1.03	4.50	1.25
n50	196.75	1.25	5.00	1.42
n100	908.75	1.31	51.75	1.46
n200	6111.00	1.07	94.75	2.26
n300	8809.00	1.02	116.50	1.90

In the second experiment, we modified the MSV-driven floorplanner developed in [20] by replacing the voltage assignment module with the proposed multicore solver. Using the same set of GSRC benchmarks, we compare the modified floorplanner with the original floorplanner. The original voltage assignment module is implemented using CS2 [8], a well-developed sequential min-cost flow solver. Performance comparison between the modified floorplanner and the original one (denoted as [20]) is given in Table II. All the results denoted as *Ours* are averaged over the 1-core (1C), 2-core (2C) and 4-core (4C) versions of our floorplanner. It can be seen that our results are very close to those in [20] on all design parameters. Especially, the power cost (P), which is the objective of the voltage assignment problem, gives almost the same results as in [20]. The tiny differences of the power cost in cases n50 to n300 are caused by the difference of total number of level shifters inserted by the two floorplanners, which consume a small amount of power themselves. With the same number of level shifters inserted, as in the cases of n10 and n30, the power cost is the same, witnessing the correctness of our multicore solver.

The running time comparisons are shown in Table III. The speedup of our program is computed against [20]. For smaller benchmark cases such as n10 to n50, the speedup is small, which is not unexpected due to the implementation overhead such as thread scheduling and maintenance of the global queue. The multi-core programs begin to gain significant speedup against original CS2 on the larger cases such as n100 to n300. Up to 2X speedup of our 4C program against [20] is achieved. We also notice the speed leap from 2C to 4C program, which shows the power of multicore programming when the number of cores increases and the parallel overhead evens out.

TABLE III
Comparison of Runtime and Speedup with Previous Work

Cases	Run Time				Speedup Rate		
	[20]	1C	2C	4C	1C	2C	4C
n10	2.28	2.82	2.44	2.57	0.81	0.93	0.89
n30	18.90	19.06	15.42	15.27	0.99	1.23	1.24
n50	55.42	70.55	55.76	49.93	0.78	0.99	1.11
n100	223.14	230.12	172.48	150.40	0.96	1.29	1.48
n200	960.31	976.86	657.21	470.01	0.98	1.46	2.04
n300	2032.86	2281.80	1625.99	1087.52	0.89	1.25	1.87

In the last experiment, we further inspect the scalability of our multicore min-cost flow solver on even larger cases, which are constructed by combining and duplicating the smaller ones. We fix the position of the blocks and run voltage assignment once instead of running the whole floorplanning because it is too time consuming to be finished by the original single-threaded program. More specifically,

TABLE II
COMPARISON OF PERFORMANCE WITH PREVIOUS WORK

Cases	Max Power (MaxP)	Power Cost with LS (P)		Power Saving (%)		Power Network Resource		LS Number		Dead Space (%)		Wire Length	
		[20]	Ours	[20]	Ours	[20]	Ours	[20]	Ours	[20]	Ours	[20]	Ours
n10	216841	167012	167012	22.98	22.98	1643	1734	9	9	6.01	6.9	6818	6866
n30	205650	142717	142717	30.6	30.6	2323	2582	37	37	13.4	13.48	32102	29636
n50	195146	145911	143562	25.23	26.43	2297	2784	48	43	16.59	16.42	67611	68003
n100	180028	126209	126442	29.89	29.77	2257	2669	106	104	14.62	15.11	126918	127447
n200	177647	133081	134091	25.09	24.52	2543	2680	165	169	16.59	16.86	228890	231119
n300	273527	171134	170232	37.43	37.76	2971	3096	146	145	24.67	23.85	254085	264317
Average	-	147677	147342	28	28	2339	2590	85	84	15.31	15.43	119404	121231
Difference	-	1	0.99	1.00	1.01	1	1.11	1	0.99	1	1.01	1	1.02

TABLE IV
SPEEDUP RATES OF MULTICORE PROGRAMS ON LARGER CASES

Cases	Constraint Graph		1C Time (s)	Speedup Rate of 2C			Speedup Rate of 4C		
	#Nodes	#Arcs		AVG.	MIN.	MAX.	AVG.	MIN.	MAX.
n200	1344	2329	0.25	1.61	1.40	1.81	2.26	1.99	2.96
n300	2209	3834	0.48	1.44	1.17	1.84	1.90	1.31	2.44
n600	4414	7662	1.15	1.46	1.26	1.60	2.24	1.87	2.64
n800	5376	9322	2.08	1.73	1.52	1.99	2.78	2.32	3.31
n900	6619	11490	1.99	1.44	1.15	1.97	2.15	1.65	2.51
n1000	6720	11653	2.45	1.76	1.51	2.02	2.92	2.36	3.30
n1200	8824	15318	3.00	1.53	1.27	1.95	2.54	2.17	3.41
n1400	9410	16319	4.20	1.83	1.67	2.03	3.16	2.86	3.44
n1600	10752	18646	4.17	1.57	1.47	1.69	2.72	2.30	3.05
Average	-	-	-	1.59	1.38	1.88	2.52	2.09	3.01

the original voltage assignment program from [20] cannot deal with cases larger than n600 due to the overflow problem. Thus, we use our single-core program (1C) as the baseline of comparison. Experiments are repeated for 10 times for each case and the average, minimum and maximum speedup rates of the 2C and 4C programs against the 1C are reported in Table IV. An average speedup rate of 2.52 and a maximum of 3.01 are achieved, showing smooth scalability of our multicore min-cost flow solver when handling large problems.

VII. CONCLUSION AND FUTURE WORK

It is desperately needed for computationally intensive CAD algorithms to speed up with the increasing number of cores in each generation of microprocessors, since their operating frequencies are largely flattened. We proposed in this paper to use nondeterministic transactional programming method to explore concurrency in algorithm design, and developed an approach to program such an algorithm on multicore platforms. By applying the method to the min-cost flow problem, one of the most important problems in CAD with tens of applications, we developed a multicore parallel program for the problem. The efficiency and the smooth scalability of the program with increasing number of cores are demonstrated in the important voltage assignment and island floorplanning problem. The design process also convinced us that such a method can be deployed to other problems. We are currently developing other multicore algorithms for CAD applications.

ACKNOWLEDGMENT

This research is supported partially by NSFC research project 60676018 and 60806013, China National Basic Research Program under the grant 2005CB321701, China National Major Science and Technology special project 2008ZX01035-001-06 during the 11th five-year plan period, the doctoral program foundation of Ministry of Education of China under 200802460068, the International Science and Technology Cooperation program foundation of Shanghai under 08510700100, the program for Outstanding Academic Leader of Shanghai, NSF under CNS-0613967, and SRC under 2007-HJ-1593.

REFERENCES

[1] R. J. Anderson and J. C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *SPAA*, 1992.

[2] B. Catanzaro, K. Keutzer, and B. Y. Su. Parallelizing CAD: A timely research agenda for EDA. In *DAC*, 2008.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[4] E. W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Commun. ACM*, 8:453–457, 1975.

[5] W. Dong, P. Li, and X. Ye. Wavepipe: Parallel transient simulation of analog and digital circuits on multi-core shared-memory machines. In *DAC*, 2008.

[6] J. F. et al. Design of the Power6^{TM}microprocessor. In *ISSCC*, 2007.

[7] U. G. et al. An 8-core 64-thread 64b power-efficient SPARC SoC. In *ISSCC*, 2007.

[8] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22:1–29, 1997.

[9] M. Herlihy. The multicore revolution. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference*, pages 1–8, 2007.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] H.Wu, D. Wong, and I.-M. Liu. Timing-constrained and voltage-island-aware voltage assignment. In *DAC*, 2006.

[13] Intel. Threading building blocks. http://www.threadingbuildingblocks .org/.

[14] D. Klingman, A. Napier, and J. Stutz. Netgen: A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Science*, 20(5):814–821, 1974.

[15] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.

[16] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Publishing Company, 2002.

[17] W.-P. Lee, H.-Y. Liu, and Y.-W. Chang. An ILP algorithm for post-floorplanning voltage-island generation considering power network planning. In *ICCAD*, 2007.

[18] C. Lin and H. Zhou. Clock skew scheduling with delay padding for prescribed skew domains. In *ASPDAC*, 2007.

[19] C. Lin, H. Zhou, and C. Chu. A revisit to floorplan optimization by lagrangian relaxation. In *ICCAD*, 2006.

[20] Q. Ma and E. F. Y. Young. Network flow-based power optimization under timing constraints in MSV-driven floorplanning. In *ICCAD*, 2008.

[21] T. Mattson and M. Wrinn. Parallel programming: Can we please get it right this time? In *DAC*, 2008.

[22] J. B. Orlin and C. Stein. Parallel algorithms for the assignment and minimum-cost flow problems.

[23] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19:279–285, May 1976.

[24] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[25] R. L. Rardin. *Optimization in Operations Research*. Prentice Hall, 1998.

[26] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Professional, 2005.

[27] X.-P. Tang, R.-Q. Tian, and D. F. Wong. Minimizing wire length in floorplanning. *IEEE Trans. on CAD*, 25(9):1744–1753, 2006.

[28] J. Wang, D. Das, and H. Zhou. Gate sizing by lagrangian relaxation revisited. In *ICCAD*, 2007.

[29] J. Wang and H. Zhou. An efficient incremental algorithm for min-area retiming. In *DAC*, 2008.

[30] X.-J. Ye, W. Dong, P. Li, and S. Nassif. MAPS: multi-algorithm parallel circuit simulation. In *ICCAD*, 2008.