On Advancing the State of Parallel CAD Algorithms

William Swartz
TimberWolf Systems, Inc.
Dallas,TX
bill_swartz@twolf.com

ABSTRACT

In this paper, we will address the the current and future of parallel algorithms and their application to CAD problems. Instance, data, and task parallelism have been applied to CAD problems with varying degrees of success. We will show how each has been applied to physical design problems and how each has fared. We will investigate the characteristics of algorithms which are most amenable to parallelism. We will also introduce the Haskell functional language and how it has been augmented to exploit data parallelism. Finally, we will present the challenges which remain for widespread application of parallel CAD algorithms.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Applications

Keywords

Parallel programming

1. INTRODUCTION

With the advent of million gate integrated circuits designs, time to market has becomes a major challenge. CAD programs, algorithms, and systems struggle to handle such large problems and their increasingly difficult constraints. The widespread adoption of 64 bit computer architectures has mitigated problems related to address space limitations. In addition, the prevalence of low cost redundant array of inexpensive disks (RAID) has all but eliminated issues related to the storage of large designs. Unfortunately, clock frequencies have not continued to scale as microprocessors have shrunk in size due to power density constraints. The result is that the execution time of a single microprocessor has saturated. In order to leverage the large silicon areas available in state of the art technologies, designers have instead turned to multiple processors on a single die. However, efficient and reliable parallel software programming techniques are in their infancy. The burden of exploiting parallel multicore microprocessors and systems has been left to the CAD programmer. In this paper, we will explore some of the current techniques available to CAD programmer.

Parallel computations may occur at the bit, instruction, data, task, and instance levels. Bit and instruction level parallelism have been exploited by the hardware manufacturer and are transparent to the CAD programmer. Techniques such as large bus widths and out-of-order instruction processing have been utilized by the monolithic microprocessor designers in order to exploit the parallelism that exists at the instruction level. These techniques are limited by the inherent parallelism found in the implementation or coding of a particular algorithm. The data dependencies of the particular coding determines an upper bound on the parallelism that can be exploited on a uniprocessor, and is typically quite modest (5 to 7 parallel instructions for the set of SPEC benchmark programs) [17].

In contrast, data, task, and instance parallelism allow programs to exploit large amounts of concurrency. In data parallelism, each processor performs the same tasks on different sets of data. There are several ways to implement data parallelism. Historically, a vector processor such as the Cray1 executes a single instruction stream or *thread* in lockstep on multiple data sets. This is known as the single instruction multiple data (SIMD) model according to Flynn [10]. More recently, the single process multiple data (SPMD) model was proposed by Pfister [8]. In the SPMD model, multiple processors using multiple independent threads work on distributed data sets using general purpose microprocessor cores. This may be implemented using message passing or barrier locks.

Task parallelism or control parallelism executes multiple instruction streams on the same or different data sets. In Flynn's taxonomy [10], this is known as the multiple instruction multiple data (MIMD) model. The data may be accessible in a single shared memory space or it may be distributed among the processing units. Each of the instruction threads is autonomous but synchronization is often required in order to scatter and gather data.

Instance parallelism is a special case of data parallelism (SPMD) where the data of the computation have no dependencies, that is, each instruction stream is completely independent and may proceed to completion without syn-

EDPS '11 Monterey, CA USA

chronization.

In this paper, we will investigate data, task, and instance parallel and how we can apply them to our CAD programs to reduce the execution time.

2. PARALLEL CAD ALGORITHMS

The remaining of this paper is organized as follows: First we will discuss the necessary features of a parallel algorithm as required by a CAD system. We then will present instance parallelism, the most successfully applied parallel CAD algorithm. Next, we will present data parallelism and show how the functional language Haskell has been augmented to automatically generate and schedule nested data parallelism. Finally, we will look task parallel and the challenges that remain for this general technique.

2.1 Requirements for CAD algorithms

As CAD tools are large and complex systems, they need to obey the tenets of systems engineering in order to be successful in the marketplace, that is, they need to be *controllable*, *observable*, and *deterministic*. A *controllable* system allows external inputs to move the internal state of the system from any initial state to any other final state in a finite time interval. An *observable* system allows the current state to be determined in finite time using only the outputs for any sequence of state and control inputs. In addition, the system must be *deterministic*, that is, the current state must be uniquely determined by current and prior states of the system.

These characteristics allow CAD vendors to properly design and test their algorithms. A CAD system must be able to regenerate a problem (debug) or recreate a result for archival purposes. Without determinism, this is impossible. In addition, non-deterministic behavior precludes the introduction of incremental design for the initial place and route must be reproducible. Hence, deterministic algorithms are a must for acceptance by the design community.

2.2 Instance Parallelism

The most successfully applied parallel CAD paradigm is instance or *trivial* parallelism. As the name suggests, each program execution is a different independent data *instance* applied to the same set of instructions. In the sequential case, the algorithm would be of the form shown in Algorithm 1.

Algorithm 1 Sequential execution

for $i = 1$ to number_of_instances **do**
  $thread_i$ ()
end for

whereas the trivially parallel form is given in Algorithm 2.

In order for the data parallelism to be trivially parallel, all variables among the threads must be independent, that is, synchronization is unnecessary or

$$\forall v_k, v_l \quad v_k \cap v_l = 0$$

where $v_k \in thread_i$ and $v_l \in thread_j$ The scheduling here is simple because

$$\forall i, j \quad t_{thread_i} \approx t_{thread_j}$$

where t_{thread_i} is the execution time for thread i. Although instance parallelism shortens the total overall execution time,

Algorithm 2 Parallel execution

$p\_iterations \leftarrow \lfloor \frac{number\_of\_instances}{P} \rfloor$
$start \leftarrow 1$
for $i = 1$ to $p\_iterations$ **do**
  $end \leftarrow start + P$ - 1
  eval_in_parallel($thread_{start}$, $thread_{end}$)
  $start \leftarrow end + 1$
end for
$remainder \leftarrow number\_of\_instances$ **mod** P
if $remainder > 0$ **then**
  eval_in_parallel($thread_{start}$, $thread_{number\_of\_instances}$)
end if

where P is the maximum number of simultaneous threads available on the computer system.

it generally doesn't scale with input size as it doesn't speed up the individual execution threads.

Let's look at a couple of examples of instance parallelism as it has been applied to CAD algorithms. Our first example is the Open Source SPICE simulator *ngspice*. In this simulator, a significant amount of execution time is spent evaluating the BSIM transistor models which are quite detailed and complex. The sequential version evaluates each instance in the inner for loop:

```
/*  loop through all the BSIM4 device models */
 for (; model != NULL; model = model->BSIM4nextModel ) {
    for (here = model->BSIM4instances; here != NULL;
         here = here->BSIM4nextInstance ){
            .
            .  /* evaluate complex model here */
            .
    }
}
```

The parallel version using the Open Multi-Processing (OpenMP) application programming interface (API) as implemented by Holger Vogt is shown below [14]:

```
InstArray = model->BSIM4InstanceArray;
#pragma omp parallel for num_threads(nthreads) private(here)
 for (idx = 0; idx < model->BSIM4InstCount; idx++) {
     here = InstArray[idx];
     good = BSIM4LoadOMP(here, ckt);
 }

 /* Load results back in to matrix */
 BSIM4LoadRhsMat(inModel, ckt);

 return good;
```

The **#pragma omp** directive instructs OpenMP to divide the instances of the model found in the array *InstArray* among the available threads and to dynamically schedule them. The implementation details are left completely to the OpenMP API reducing the burden on the programmer.

Originally, the evaluation of the transistor models took 62% of the execution time. From Amdahl's law [2],

$$\frac{1}{r_s + \frac{r_p}{n}}$$

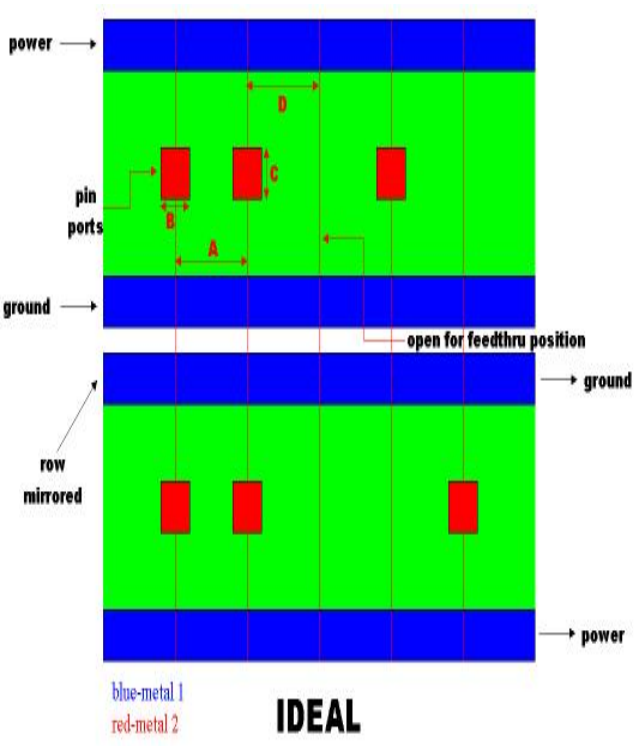

Figure 1: Ideal standard cell layout for decomposition.

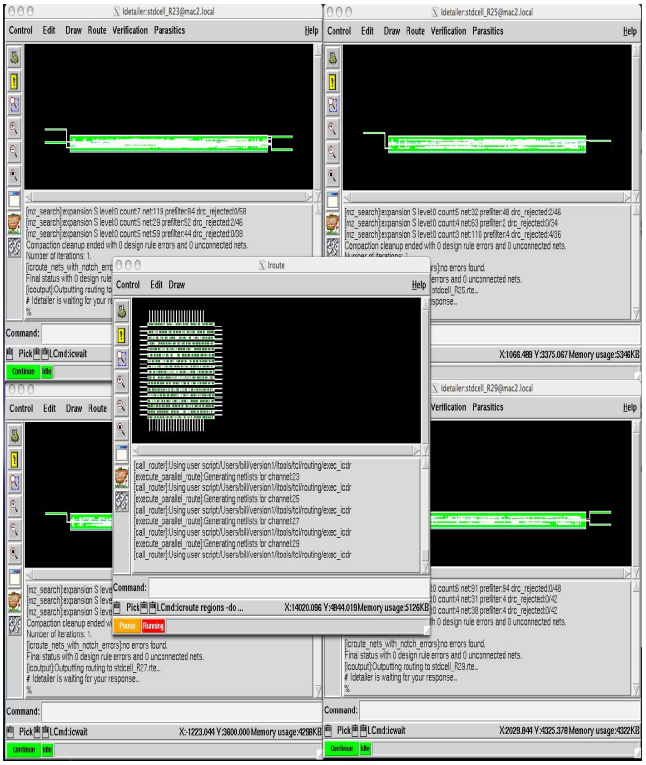

Figure 2: Parallel routing after partitioning is simple case of instance parallelism

we can expect this enhancement to give us a maximum speedup of 2.63 as the number of processes goes to infinity and in practice it was found to yield a speedup of 2.01 when four threads were available.

Another successfully applied algorithmic technique is to use geometric partitioning to induce instance parallelism. For example, a global router may partition a row-based design such that all subsequent detail routing is completely independent. This is possible under ideal conditions, that is, the standard cell design methodology follows the design template found in Figure 1. The most important features for decomposition are that the pin's ports must align in the center of the row and that at most one pin resides per column. This allows the global router to create virtual regions between adjacent rows of the center pins. Each of these regions are independent as the global router has determined the routes between the virtual regions effectively partitioning the routes among the different virtual regions. The partitioning allows each virtual region to be treated independently and become *trivially* parallel. In Figure 2 we can see four independent regions being routed simultaneously while the center display shows the entire design in the global router.

2.3 Data Parallelism

Data parallelism exploits the need for programs to perform bulk data options. Traditionally, data parallelism was applied only to *flat* data or data that can be referenced from a single array. These arrays could be interpreted as *vectors* and computer architectures were tailored to facilitate efficient operations on these vectors [15]. However, all sequential operations on the bulk data need to be of a similar execution time in order to minimize *dead time*.

In the early 1990s, Blelloch and others developed the idea of nested data parallelism and the NESL programming language [5]. Nested data parallelism generalizes the bulk data to recursive and unbalanced data structures. Nested data parallelism greatly expands the applicability of data parallelism for it operates on all levels of hierarchy within an algorithm. Such flexibility allows a much more diverse set of applications such as sparse arrays, multilevel adaptive grids, divide and conquer algorrithms, graph algorithms (including shortest paths and spanning trees), and graphics.

Recently, the Haskell functional programming has been augmented to support nested data parallelism. Unlike conventional imperative languages such as C and C++, the Haskell language is a pure functional language without side effects. Without side effects, Haskell greatly improves observability and controllability of programs. In addition, the compiler can safely make assumptions not possible with imperative languages and this greatly expands the number of safe optimizations possible. The key idea behind Blelloch's work is a transformation from nested data parallelism to flat data parallelism which is possible for functional languages. Once transformed into flat data parallelism, efficient dynamic scheduling can be performed. The procedure is completely deterministic. The Haskell compilation process consists of four steps [11]: 1) Desugaring which removes syntactic sugar, reducing the program to a simple lambda language. This intermediate language, GHC's "*Core*" language, is still strongly typed. 2) Vectorization which transforms nested data parallelism into flat data parallelism within the

Core language. 3) Fusion of structures which optimizes the program by eliminating redundant synchronization points and intermediate arrays, thus dramatically improves locality of reference; and 4) Gang parallelism which divides the parallel operations spatially into chunks, each chunk being executed by a thread from a gang of threads. Typically a gang contains a thread for each CPU. Gang parallelism is expressed by giving library implementations of the "vector instructions", rather than by built-in compiler support. The results reported on shared memory architectures by these researchers are encouraging as their work scaled well up to 8 cores [12]. We are currently implementing a placement algorithm using this methodology.

2.4 Task Parallelism

The most general form of parallelism is task parallelism where programs are divided into tasks and distributed among the processors. The challenge is to distribute the tasks such that synchronization time doesn't dominate the runtime. Historically, the design of task parallelism algorithms was a manual and intensive task, effectively limiting the number of parallel CAD applications available.

For example, parallel algorithms for simulated annealing which produce high quality results have been long sought due to its long run times. Simulated annealing (SA) is a general stochastic iterative improvement heuristic which optimizes cost functions in a large search space [13]. The simulated annealing algorithm randomly searches nearby states in the solution space by using the Boltzmann probability function to allow uphill moves. A global parameter T called the temperature controls the size of the uphill move. The temperature T is varied from a high value to zero as the algorithm progresses. This allows the algorithm to avoid being stuck at local algorithms. In fact, given enough time the algorithm will converge to the global optimum. Unfortunately, the temperature must be lowered slowly to achieve such a high quality result, that is, the algorithm must visit many nearby solutions.

It is clear that a parallel simulated annealing algorithm would be of great use in CAD. For example, a simulated annealing placer is a great candidate for parallel execution. Unfortunately, several issues make parallel implementations problematic. Typically, the cost function of a placer will involve wire length and overlap terms. The wire length costs require large amounts of communication as any cell could have an external connection outside its partition. At high temperatures, cells move long distances across the integrated circuit area and this results in many synchronizing steps. The locks and barriers to provide proper synchronization greatly increase the overall run time of the program and thereby limit the effectiveness of the parallel algorithm. However, at low temperatures cells are more likely to remain local to the partition and therefore communication can be minimized. Low temperature simulated annealing has an efficient parallel implementation scaled and so the high temperature regime becomes the time bottleneck. At this time, no scalable and efficient simulated annealing algorithm based on task parallelism has been presented.

in Figure 3, we see a simulated annealing placer at high temperature. One can observe that the rows are unequal due to assumptions made by each processor. In the implementation, each processor makes the assumption that the row length will be independent of the moves made by other

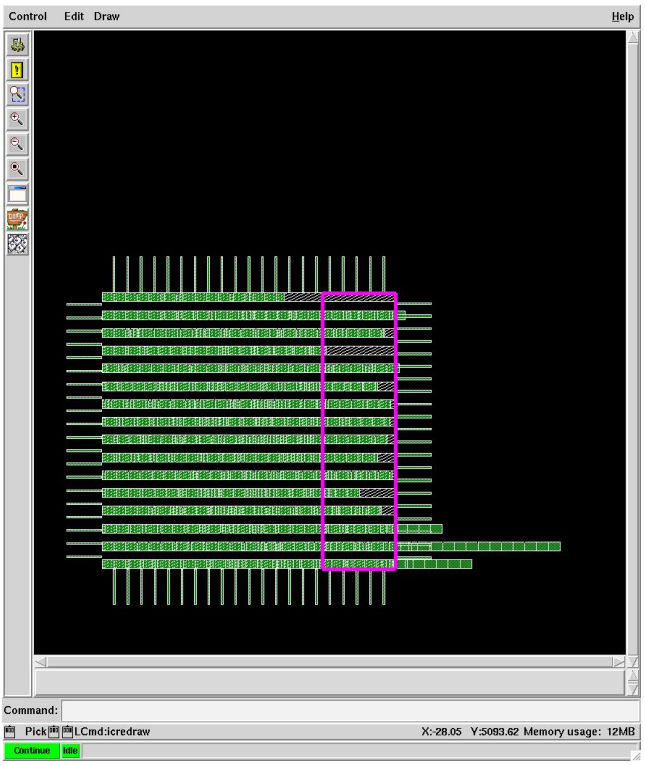

Figure 3: Simulated annealing parallel placement using geometrical partitioning. Shown at high temperature, lack of synchronization causes row length unevenness.

processors. As the number of partitions increase, this assumption becomes increasingly invalid. The solution is to synchronize the processors with information about the row lengths of other processors. However, this synchronization causes a major bottleneck because of the frequent updates that must be performed.

2.4.1 *New Tools for Task Parallelism*

Recently, tools such as Cilk/Cilk++ [7] [1] have been introduced to reduce coding time (by automatically generating and dynamically scheduling the tasks) and to improve the quality of the result by detecting race conditions between synchronized variables. In addition, various libraries and concurrent languages have been proposed to ease the burden on programmers.

One of the most promising deterministic approaches is *Precision Timed C* (PRET-C) methodology [3]. PRET-C uses a 4 stage process to insure determinism in a shared memory environment. In the first stage, the C language is extended using a small set of macros. Next, these macros are converted into an intermediate format which allow easy construction of a finite state machine. In the third stage, the finite state machine is augmented with execution costs. Finally, the worst case times are generated based on the set of timed finite state machines and obey rules to maintain safe synchronization.

2.4.2 *Observations*

Empirically, it can be observed [4] that problems where

progress has been made towards viable task parallel algorithms are those problems which exhibit *power* or fractal law distribution of communication requirements, that is, $p(x) \propto L(x)x^{-\alpha}$ where $\alpha > 1$ and $L(x)$ is any function satisfying $\lim_{x\to\infty} L(tx)/L(x) = 1$ and t is a constant. Any problem requiring more communication has failed to be efficient like the previously mentioned high temperature annealing regime.

Many problems that have been mapped successfully to *graphical processor units* (GPUs) can be geometrically partitioned. Problems such as molecular modeling [16] and temperature analysis have natural partitions. In addition, the calculations tend to be local lowering the communication requirements. However, most CAD problems have exploited data rather than task parallelism in their application to GPUs [9] [6].

From these observations, it can be conjectured that all problems that can be partitioned geometrically with a fractal distribution of communication have an efficient and scalable task parallel implementation.

3. CONCLUSIONS

In this paper, we presented various techniques that have been applied to improve the state of the art in parallel CAD algorithms. We have shown instance, data, and task parallelism in algorithms and CAD algorithms be might beneficial in their use. Most of the promising techniques are limited to shared memory architectures whose scalability is still in doubt. Much work needs to be done to offer a general and flexible methodology for the incorporation of parallel algorithms in to CAD applications.

4. REFERENCES

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, GA, USA, Apr. 2010.

[2] G. Amdahl. Valitity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, April 1967.

[3] S. Andalam, P. S. Roop, and A. Girault. Deterministic, predictable and light-weight multithreading using pret-c. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1653–1656, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[4] G. B. Bezerra, S. Forrest, M. Forrest, A. Davis, and P. Zarkesh-Ha. Modeling noc traffic locality and energy consumption with rent's communication probability distribution. In *Proceedings of the 12th ACM/IEEE international workshop on System level interconnect prediction*, SLIP '10, pages 3–8, New York, NY, USA, 2010. ACM.

[5] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of Principles and Practices of Parllel Programming*, pages 102–111, 1993.

[6] J. Cong and Y. Zou. Parallel multi-level analytical global placement on graphics processing units. In *ICCAD*, pages 681–688, 2009.

[7] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, Oct. 2005.

[8] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.

[9] Y. S. Deng, B. D. Wang, and S. Mu. Taming irregular eda applications on gpus. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, New York, NY, USA, 2009. ACM.

[10] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948, 1972.

[11] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 383–414, December 2008.

[12] G. Keller, M. M. T. Chakravarty, R. Leshchinskiy, S. L. P. Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *ICFP*, pages 261–272, 2010.

[13] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[14] R.-K. Perng, T.-H. Weng, and K.-C. Li. On performance enhancement of circuit simulation using multithreaded techniques. In *International Conference on Computational Science and Engineering*, pages 158–165, Aug 2009.

[15] R. M. Russell. The cray-1 computer system. *Communications of the ACM*, pages 63–72, 1979.

[16] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[17] D. Wall. Limits of intruction-level parallelism. In *ASPLOS-IV Proceedings of fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, April 1991.