

Parallel Pattern Matcher

CS267 Final Project

Frank Gennari, Shariq Rizvi, and Guille Diez-Canas

1. Introduction

As the critical dimension in optical lithography continues to shrink and integrated circuit masks become more complex, it is becoming more important to determine where the layout is affected the most by non-ideal process conditions. If the most problematic shapes can be identified and represented in a simple way, then these test patterns can be used to locate areas in any layout that are the most sensitive to these process effects. After the ‘hot spots’ have been found, the designer can use these results to go back and alter the geometry to reduce the sensitivity to these effects. This procedure will reduce the effects of imperfect optics on the printing of the image from the mask to the wafer, improving the yield of the design. Locations of interest can be filtered to include only points along edges, line ends, inside corners, and/or outside corners. Additional information on the pattern matching theory and validation¹, applications of pattern matching², and pattern matching algorithms³ is available from SPIE and ISQED. Some of the pattern matching concepts and algorithms are similar to those used in Optical Proximity Correction (OPC)⁴.

The general pattern matching method presented here can be used to search for matches between any user-defined bitmap image and any set of input polygons on multiple layers involving complex floating-point number weights. A typical 128 by 128 pixel pattern yields reasonable speed vs. accuracy and is scaled to represent a radius of several wavelengths. Examples of pattern matching runs involving lithographic lens aberration patterns matched to phase-shifting masks are shown in Figure 1 and Figure 2. The pattern matching algorithms have been optimized for efficient processing of full chip layouts, such as the one shown in Figure 3.

Frank Gennari has been working on the pattern matching system as Ph.D. research for the past three years. The goal of this paper is not to describe the pattern matcher work in great detail but to explain how we converted a large sequential program into a scalable parallel program.

2. Software System

A 2D pattern-matching algorithm has been developed to scan over the layout and compute, for each point of interest, the correlation of the actual local layout geometry to the shape of the test patterns. The goal of the software is to find and rank order the locations of the worst-case effects in a full chip layout based on the highest match factors. A block diagram of the pattern matcher is shown in Figure 4. The pattern generator reads a set of user-defined equations in order to generate the pattern bitmap. The main pattern matcher executable then reads the pattern, a user input parameter file, and a mask layout in GDSII (possibly zipped) or other standard layout format. The pattern matcher is capable of reading these layouts directly, storing them in compact hierarchical form for efficient access to geometry. The pattern matcher is run, and a resulting

sorted table of the highest match factors and their locations for each pattern is output as well as JPEG images of the patterns over the match locations.

The pattern matching software also provides a graphical interface for the user to view the layout with the highest scoring match locations highlighted. The non-graphical part of the system and some of the graphical parts are cross-platform and have been compiled under Windows, Solaris, and Linux on various 32- and 64-bit architectures. The vast majority of the code, programming effort, and runtime are spent in the core pattern-matching executable, highlighted in red in Figure 4. Thus this component of the system was chosen to be parallelized.

3. Algorithms

The matching algorithm itself has three main steps. First, the input polygons are split into rectangles and right triangles. Next, the rectangles and triangles are spatially subdivided and sorted to permit efficient access to local areas of data. This includes partitioning the layout into smaller overlapping areas and locally flattening any hierarchy. This partitioning step was convenient in that the partitions are independent and can be processed in parallel, so this step was the point at which the work was divided among processors. Although this part of the code is embarrassingly parallel, there were a number of issues to deal with.

The third step is the matching loop itself, where all of the runtime is usually spent. The number of points at which the match factor is computed depends on whether the user is searching all or part of the layout and if the match locations are specified to be corners, edges, line ends, or all points on a layout grid. Even if the points are constrained to lie on the edges of features, the number of test locations can be in the billions for a layout of several square centimeters. Therefore, it is critical to minimize the time taken to compute the match factor at a point. The pattern-matching algorithm relies on the fact that the match factor is a linear sum of independent contributions from the weights of each of the primitives overlapping the pattern.

Complex input layout shapes are difficult to store, access, and process quickly inside of the matching loop, and for this reason they are split into simpler rectangles and right triangles as a preprocessing step. The overall goal of the polygon splitting algorithm is to produce a minimal, non-overlapping set of smaller primitives that together cover the entire area inside the polygon and none of the area outside the polygon. Most of the polygons found in an integrated circuit mask layout are Manhattan and can easily be split into rectangles. However, the occasional non-Manhattan polygon must be split into right triangles as well as rectangles if possible in order to avoid the large numbers of small rectangles required to approximate the diagonal edges.

The simplest way to compute the match factor at each location is to perform a pixel-by-pixel correlation computation between the pattern and the shapes in the layout that overlap the pattern. This algorithm is extremely slow and scales as the area of the pattern times the number of locations in the layout. The trick to speeding up the match factor computation is to avoid touching each of the many pixels in the pattern. This can be accomplished by pre-integrating the pattern in two dimensions and then adding contributions from rectangles and triangles based on table lookups of their vertices³. The time taken to compute the match factor at a location is now proportional to the number of rectangles and triangles overlapping the pattern, which is much smaller than the number of pixels in the pattern. This leads to two orders of magnitude of speedup.

The inner loop that iterates over the shapes overlapping the pattern involves integer operations for coordinate clipping, floating point operations to scale and sum the weights of the shapes, and memory lookups into the pre-integration matrix. The time taken in each of these operations depends on the machine architecture, but in any case about 60% of the total runtime is spent here. Therefore at least this part of the algorithm must be parallelized for good performance.

4. Sequential Performance

The pattern-matching engine has been highly optimized for processing large input files and, in general, the geometry processing algorithms are competitive with commercial Electronic Design Automation (EDA) tools. The rectangle-based matching algorithm was tested on a number of large layouts in various technologies with one or more layers. In all cases, the actual match time was proportional to the number of rectangles in the flattened layout times the layout density. The largest test layout consisted of the 17MB GDSII file of the active area mask layout of a microprocessor with area 417mm², eleven levels of hierarchy, and 35.3 million flat rectangles. The test pattern was a 128x128 bitmap with pixel size of 100nm, and the layout contained 2.6 billion potential edge match points, though only about 20% of these points were tested due to an adaptive filtering algorithm. The match took approximately 70MB of memory, most of which was used to store the rectangles. Table 1 shows sequential pattern matching runtimes on various processors.

Processor	Runtime (min.)
1GHz PIII / Windows 2000	34
3.2GHz P4 Xeon / RH Linux	15
1.3GHz Itanium2 / Linux	25
Seaborg Node (IBM Power3) / AIX	150

The Itanium2 machines did not perform as well as expected since this input file contained only rectangles on a single layer. The Itanium2 performed much better on multi-layer layouts and layouts with diagonal edges since the multilayer case involves more floating-point multiplies and triangles involve lookups into larger pre-integration tables and benefit from a large cache. The Seaborg machine did not perform well, possibly because it is extremely slow at the integer comparisons involved in testing which rectangles overlap the pattern, and because memory allocation of small sizes apparently takes longer than on other machines.

The actual steps of the sequential pattern-matching algorithm are shown in Figure 5. The left side of the figure lists the steps and the color code used in this figure and in Figure 6. The right side of Figure 5 shows the breakdown of runtime, approximately to scale for typical inputs. The actual matching step takes about half the runtime, and is fortunately easy to parallelize. The database operations take about 25% of the runtime and can be parallelized with more difficulty. The input file read takes a significant part of the runtime but could not be completely parallelized, as explained in section 5.3. Finally, the other steps are very fast and are probably not worth parallelizing.

5. Parallelization Strategy

Figure 6 explains how the parallelization was accomplished. The colors correspond to the color code in Figure 5, and the heights of the blocks correspond to approximate runtimes of those steps. The parallel algorithm involves spatially subdividing the layout in both the X and Y directions and processing the partitions in parallel. The layout is already partitioned to conserve memory, but parallelization of the partitions is not easy. The partitioning is somewhat complex since the partitions must overlap and load balancing may be difficult. There does not appear to be a way to load the compressed hierarchical layout database in parallel or efficiently distribute the storage due to complex dependencies and overlaps between the cells. Thus each node must load its own copy of the hierarchical database.

The current algorithm uses an approximation method that is based on previously computed statistics, and it was initially thought that these values would have to be communicated between the processes at various intervals in order to effectively use the approximation algorithm. However, the approximation algorithm reaches a steady state after a few seconds, so each processor will independently reach the same value very quickly. The problem became much easier now that this communication was unnecessary.

Since the computation was expected to dominate over the communication, we decided it should work well on a distributed memory system using MPI. The program is in C++, but most of the inner loops where performance is critical are really in C. There are several modules in the software system totaling about 32,000 lines excluding the interactive GUI code, but the part to be parallelized is much less. Most of the code blocks were untouched since the new parallel code went into the spatial subdivision algorithms and the outer loops. However, a significant amount of code change was required in order to make it thread-safe.

5.1 Spatial Subdivision for Parallelization

We decided to use two parallelization steps. First, each node loads the entire database and spatially partitions it into several hundred areas, and node N will process every partition P where $(P \% \text{Numnodes}) == N$. Then, the partition is further subdivided into regions, where each processor in node N will work on its own set of regions. MPI was used for the inter-node communication, and pthreads was used for communication within each node. This partitioning strategy is illustrated graphically in Figure 7 for the combined MPI and pthreads case, with four nodes each containing four processors. The actual time taken to process a partition is highly variable due to the non-uniform distribution of polygons in an integrated circuit layout. However, if each node and each processor are assigned a large number of partitions evenly distributed over the entire chip area, then the overall variance per processor will be small. This leads to good load balancing across symmetric processing elements when the number of partitions is much larger than the number of processors.

This use of pthreads will be referred to as the “inner” loop version. The problem with the inner loop version is that the process of further subdividing the partition into sub-regions involves processing that takes as much as 10% of the runtime. An alternative method that was implemented, named the “outer” loop version, involves distributing the partitions among processors using a round-robin algorithm similar to the node partitioning method. The disadvantage of the outer loop method was that it required several partitions to be in memory at

once, one for each processor. This was not a problem in our test examples though since both Citris and Seaborg had adequate memory for these small test cases. The largest test run required only 577MB of RAM.

5.2 Merging of Results

Initially, all processors loaded the entire layout database, ran on their partitions, and sent their results to processor 0. Sending all of the results to processor zero will be inefficient when the user asks for a large number of results, so this method was replaced with a binary tree-based results merge. A simple four-processor case of this binary tree is shown in Figure 8. In the first phase, node 1 sends its results data to node 0 for merging, and at the same time node 3 sends its data to node 2. In the next merge level, node 2 sends its data to node 0, and then node 0 outputs the final results. Figure 9 is a merge tree diagram for a larger number of nodes. This binary tree merge algorithm worked well even for 32 processors (5 levels), with a merge time of less than a second.

5.3 Parallel Pipelined File Read

Once the core pattern matching algorithm is embarrassingly parallelized, the time taken to load the whole layout on each node, unzip it, and pre-process becomes more dominant. Given that pre-processing does not require the complete uncompressed layout to be in memory, there is scope for parallelization between the read/unzip and pre-process tasks.

We parallelized these two operations by dividing the layout file into 8MB chunks and pipelining these chunks between read/unzip and building the hierarchical database (pre-processing). Specifically, pthreads were used to spawn two different threads that perform these tasks in parallel. The algorithm proceeds as follows:

1. Allocate two blocks of memory both equal to the size of a chunk plus the largest possible record size of 64KB (records are discrete elements inside a GDSII file).
2. Read chunk 1 into block1
3. $i=1$
4. Read chunk $i + 1$ (if there is more data to read) into block2 and pre-process chunk i (which is in block1) in parallel
5. Synchronization operations; May need to move a partial record from block1 to block2
6. Swap pointers to block1 and block2
7. $i=i+1$
8. If there is data to process, go to 4

The reason each block has to be larger than the size of a chunk (step 1) is because the last record inside a chunk may be incomplete, and may have to be appended in front of the next chunk before it is pre-processed (step 5). In such a case, the block should be capable of holding data equal to a chunk plus this incomplete record.

The preprocessing has not been parallelized and large input files still have a large sequential load time. The load time was reduced up to a factor of around 1.7, depending on the

layout, whether or not it was zipped, whether or not the file was cached locally, the state of the network, and the number of nodes loading the file at the same time.

Given that the read/unzip task is typically much slower than the pre-process task for a chunk, we did not find the parallelization very effective. The next step was to parallelize the read/unzip tasks. An initial attempt tried to read and unzip in parallel, in a pipelined fashion. However, unzipping is inherently a sequential task, and the interface exported by the ZLIB library did not permit any easy way to achieve this. The disk I/O time is also sequential, so we could not achieve significant parallelism in the load phase.

5.4 Dynamic Load Balancing Queue

The partitioning scheme described above statically assigns work to processors. All processors know in advance the work they must do, and no communication is needed to split work. This approach has the drawback of not optimally balancing the load in heterogeneous systems. When using slow (900 MHz) and fast (1.3 GHz) nodes on the CITRIS cluster, or when some of the nodes assign a fraction of the CPU time to the parallel pattern matcher (due to CPU sharing with other jobs), the faster nodes finish their assigned tasks much sooner than the slower ones. This fact is seen when performing the final merge, when all processors report their matching results, and results in a potentially large wait time synchronizing the processes before the merge.

In order to eliminate this wait time we implement dynamic load balancing. Since the mapping from partitions to be processed onto the processors changes with time, a server thread is set up that handles the dynamic queue. Nodes run a client that communicates with the server through MPI to ask for a partition to process. The server itself is running as a thread on processor zero. It has a queue of partitions and hands them one by one to clients on demand. However, the server thread is busy a small fraction of the time, spending most of its time waiting for an incoming request from any of the clients. This means that it makes sense to run another thread on processor zero, doing actual work, which in turn will also have to communicate with the server.

The above scheme is simple and effective, with all clients sending requests to the partition server whenever they need more work to process. We found a complication with this approach. Node zero runs both the server and, in order to better utilize the processor, also another client thread that sends MPI requests to the server. MPI, however, seems to have problems using various types of communication running on different threads on the same node, and working this out was not possible without a change in design. Possible approaches where to make node one's client acts as intermediary of node zero's client's requests. This results in moving the problem to other nodes, where this time node one would need a client and a server thread, both of which use MPI independently. Finally the cleanest solution we found was to use two types of communication. Since MPI is problematic within node zero, MPI is used only for communication between the server and clients at other nodes, while the client at node zero communicates with the server through shared memory. This is not an entirely clean approach, since one client is effectively reading and writing the state of the server in order to serve itself. However this is done safely through the use of locking mutex constructs to ensure the consistency of the server state.

The final scheme we implemented has the form seen in Figure 10. Each node has one client, with the exception of node zero, which also has a server thread running. Communication

is done through MPI between all clients and the server, which exchange only two short messages, one from the client letting the server know it needs more work, and one from the server to send the identifier for the piece of work to be done. Termination is signaled by the server sending a magic identifier to the client, at which point the client understands that there isn't any more work to do. The exception to this mode of communication is the client of node zero; this client can see the server state (which has to be in the global scope), and itself reads and modifies the server state in order to obtain a piece of work to do.

An interesting extension to this happens when several threads are running on each node, each one acting as a client and sending requests to the server. We have tested this and it has worked without changes to the original implementation. The node that has the server thread has no particular problem since all the client threads in the node access the server state through mutexes and therefore this is extensible to any number of client threads in the node. The remaining nodes have no server but several client threads, each sending requests to the server using MPI. The interesting thing to note is that no matter what order the answer to the requests are received, every thread will always receive an answer to its query. The only unexpected result is that threads receive answers out of order, but since there are no ordering dependencies in the partitions to be processed, this poses no problem and works out fine without modifications.

Finally, we make a note about the choice of granularity. Obviously, the larger the units of work that are dispatched by the server, the smaller the overhead of communicating with the server. However, distributing large pieces of work means that when the work is completed there will be a higher wait time when synchronizing all the processes. We choose the granularity of the load to be at the partition level. Partitions have varying complexity, but in most cases tested on the CITRIS cluster the average processing time for a single partition was in the order of one second. This seems reasonable to us since one second is enough time to amortize the costs of communication with the server. In the cases we tested, the communication time with the server amounted to below 1% of the total processing time. At the same time, this choice of granularity is also small enough to keep the synchronization wait time before the final merge quite small.

6. Results

6.1 MPI Performance

The MPI results for the Citris cluster are shown in Figure 11. The parallel pattern matcher has a high parallel efficiency, achieving over 28X speedup on 32 nodes for the 25 min. test run, excluding the MPI startup time. The runtime was reduced from 25 minutes to only 51 seconds. The match time is near the theoretical limit, though there is a large MPI startup time of several minutes when using many nodes. This MPI startup time will not be as much of an issue when running on larger files that take hours to run sequentially and tens of minutes in parallel. This speedup was actually achieved with only one processor per node, which does not utilize all of the processing resources. However, the matching can also be partitioned using pthreads, and a 16 node run using both processors (two threads) achieves a speedup of over 29. It appears as though pure MPI and MPI + pthreads work equally well, but MPI + pthreads is preferable because it uses all available processors and less memory (only one database per node). The reason the speedup is not perfect is due to several issues, most notably the fact that the layout load and

preprocessing is not done in parallel and has a small constant time penalty regardless of the number of processors.

Figure 12 shows the MPI speedup on Seaborg. These numbers are for a single node and multiple MPI tasks per node. It was found that MPI worked better than pthreads on Seaborg nodes, but this is due to pthread memory allocation problems as discussed in the next section. Seaborg had much longer runtimes than Citris on the standard benchmark input file, and could not be run in debug mode. Due to time limitations, we decided to use a smaller example on Seaborg to get more reasonable runtimes. We also had to choose an example that used less than 128MB of memory so that we could speed things up by running interactively. The MPI scaling for Seaborg is shown in Figure 12, for two different configurations. The first configuration involved starting an MPI job that ran on multiple nodes, using only one processor per node. The scaling was still quite good, giving a speedup of over 11 for 16 nodes, but it is not as good as on Citris due to the increased ratio of preprocessing time to match time required for this smaller example. In this case, significant time was spent in the pattern pre-integration, which has not been parallelized. We did not have time to parallelize the pre-integration, and anyway a much faster pre-integration algorithm is under development that should not need parallelization. The second configuration involved running MPI on the processors within a single node. This configuration did not scale as well due to memory contention within the node and the time taken for a single node to load multiple copies of the file from disk. The I/O problem could have been avoided by reading from a zip file to reduce file read time, but we could not get zlib to compile on Seaborg.

6.2 Pthreads Performance

We attempted to characterize the performance of pthreads on various dual processor systems in order to determine what the scaling limitations were. Unfortunately, we were unable to achieve good performance with pthreads on Seaborg, as the runtime actually increased with the number of threads. We believe that the threads are fighting with the memory system when allocating small amounts of memory inside of the STL vector resize (doubling array), which occurs when constructing partial vertex arrays inside of the recursive polygon splitting code. It is not clear how to remedy this problem, since we are not familiar with the vector allocators available with the version of the STL installed on Seaborg. Given more time this problem could likely be solved.

Figure 13 shows the relative performance advantages that come from utilizing both processors of a dual processor machine. As explained in the previous section, MPI can be run on both processors to achieve a near perfect speedup of 1.95 on a Citris Itanium2 node. This is because each processor builds the entire database and there is no memory sharing. If pthreads are used instead of MPI, then the speedup is slightly lower due to memory contention and time spent waiting at mutex locks. However, the memory requirements are lower since only one copy of the database is stored, though two partitions are also stored. If the pthread split is done later in the code (inner loop) at finer sub-region granularity, the memory is reduced further but the speedup falls since a smaller portion of the processing is done in parallel. The database access is still sequential.

A 3.2GHz P4 Xeon with hyperthreading was also tested. Both pthread algorithms resulted in a speedup of less than 1.5 due to memory contention. Memory bandwidth is more of a

bottleneck on the Xeon system than on an Itanium2 since the processing speed to memory bandwidth ratio of the Xeon is so much higher. Starting four threads on the Xeon instead of two results in an additional speedup, bringing the total speedup to 2.25. This extra speedup comes from the hyperthreading technology in the Xeon processor, which allows a single processor to run two threads at once. Hyperthreading is not as good as having two real processors, but it does help somewhat. The Xeon is very fast at sorting the integer data in the database operations, and doesn't suffer from much slowdown when using the inner pthreads split instead of the outer split.

6.3 Dynamic Queue Results

We tested the impact of dynamic load balancing on the performance of the system. The objective is to eliminate, to the extent possible, the wait time in the synchronization between processes before the final merge of the results. This is shown in Figures 14 and 15. The system was run several times for each case and the results were averaged to reduce variance, which, when running on the CITRIS cluster, can be an issue. Results from running the system without load balancing are shown in Figure 14 (load and unzip time is not included). This figure shows the runtime for an execution of the system on eight processors, some of which are slow CITRIS nodes (900 MHz) and some of which are fast (1.3 GHz). The horizontal axis represents the number of slow nodes among the eight running. As expected, the runtime for the actual partition processing (blue bar) increases slowly as more slow nodes are inserted. The wait time to synchronize before the final merge increases sharply from near zero and quickly reaches a point where using more or less slow nodes does not affect the total runtime because fast nodes always have to wait for slow ones anyway. Note that in Figure 14 the communication time is zero because the static algorithm requires no communication. An interesting effect occurs when using eight fast nodes. Because all nodes are fast the load balancing should be quite good in this case, but repeated testing of this case yielded approximately the same high load imbalance. This may be due to the fact that processor load on the CITRIS cluster is variable and usually at least one of the fast (and therefore more heavily used) nodes can be expected to not run at peak speed. This imbalance produces the same effect as having some slow nodes: increased wait time before the merge. This further justifies the utility of balancing load, since in many practical situations, not only the heterogeneity of the system produces imbalance, but also the fact that computational resources are often shared.

The result of running the load-balancing scheme we describe in section 5.4 is shown in Figure 15. The wait time before the final merge has not completely disappeared but is almost entirely gone. This improvement comes at the cost of some communication overhead, shown in magenta. The communication cost is very small in all cases, with a maximum at 1% of the total time and a minimum of around 0.2% for all tests that we run. Finally, the wait time cannot be completely eliminated, and is related to the granularity of the pieces of work that the dynamic load balancer dispatches. We chose to set the granularity at the partition level, with a runtime of approximately one second each partition. This means that the wait time to synchronize processes at the end of the partition processing will not exceed roughly one second, and is in practice smaller than this number on average.

7. Conclusions

Overall, the attempts we made to parallelize the pattern matcher were successful. It was surprisingly easy to get a speedup of 28 for one example running on Citris, but that was the end of the easy part of this project. Making the entire pattern-matching algorithm after the partition split thread-safe involved changing a great deal of code and was no quick task. As expected, we did not achieve much parallel speedup with the zip file load, but we did find a way to store the statistics resulting from the first pass of the GDSII file read so that it only had to be read and unzipped once after the first run. The dynamic load-balancing queue was difficult to write due to problems we encountered when combining MPI and pthreads, but in the end we found an effective solution to the load-balancing problem.

We encountered a number of problems on Seaborg, which limited the data we could extract from that system. Given the correct documentation and some additional time, we would likely have solved the pthread and zlib problems. Seaborg is a unique system and requires some additional work when porting large applications.

The parallel pattern matching engine is much faster than it was initially expected to be, and scales surprisingly well on eight or more processors even for runs that take only a few seconds. There are still areas that need work, such as parallel pattern pre-integration and parallel results processing, but the pattern matcher is well on its way to becoming the next big EDA tool.

References

1. F. Gennari, G. Robins, and A. Neureuther, "Validation of the Aberration Pattern-Matching OPC Process," SPIE Vol. 4692B, 3/02.
2. A. Neureuther, F. Gennari, "No-Fault Assurance: Linking Fast Process CAD and EDA," SPIE Vol. 4889, 10/02.
3. F. Gennari, A. Neureuther, "A Pattern Matching System for Linking TCAD and EDA", ISQED 3/04.
4. Nick Cobb, "Fast Optical and Process Proximity Correction Algorithms for Integrated Circuit Manufacturing," PhD Thesis, University of California, Berkeley, 1998, pp. 38-45.

Figures

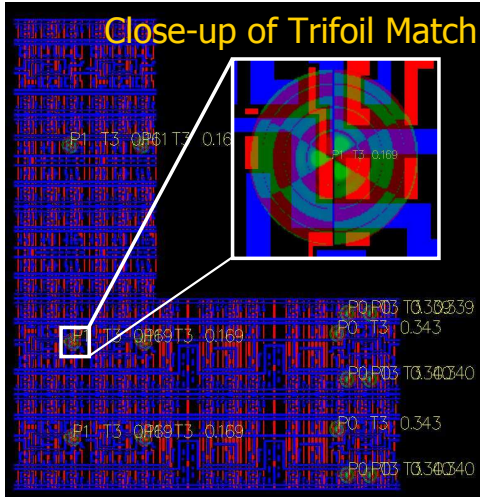


Figure 1: Example of trifoil lens aberration match.

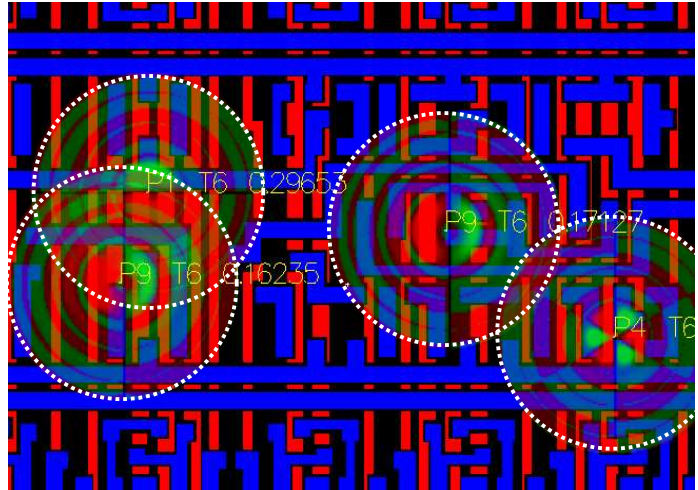


Figure 2: Example pattern matches for various lens aberrations.

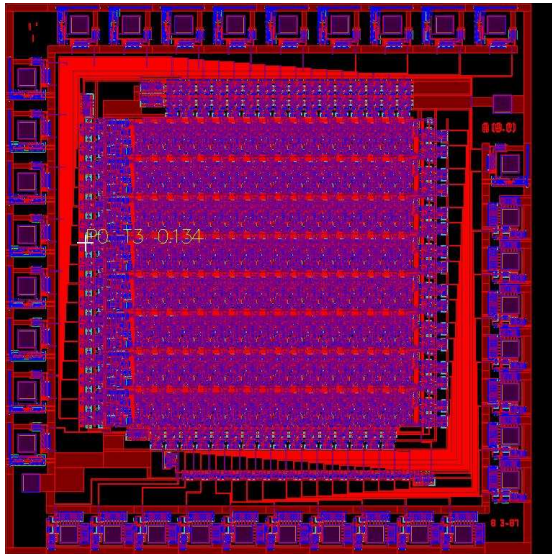


Figure 3: Example full chip layout used for pattern matching.

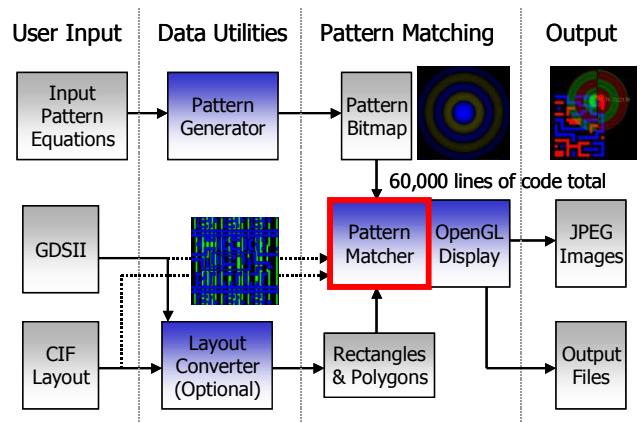


Figure 4: System block diagram, with parallel code highlighted in red.

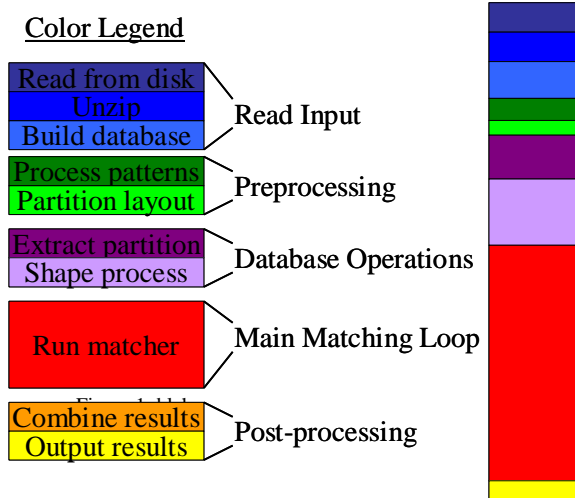


Figure 5: Pattern matching algorithm steps with approximate runtime distribution.

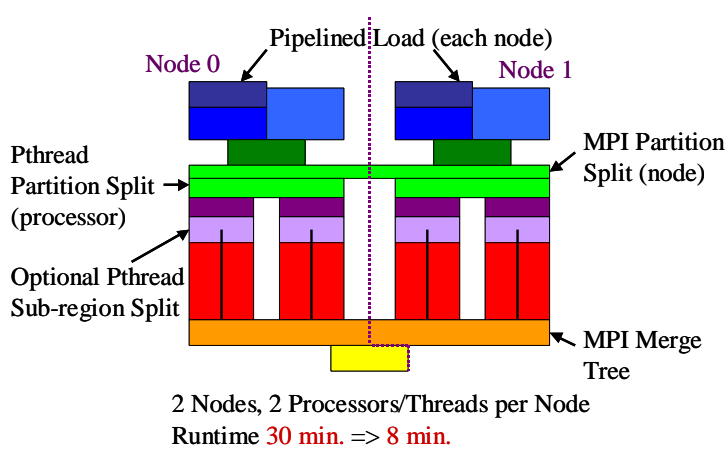


Figure 6: Pattern matching parallelization strategy.

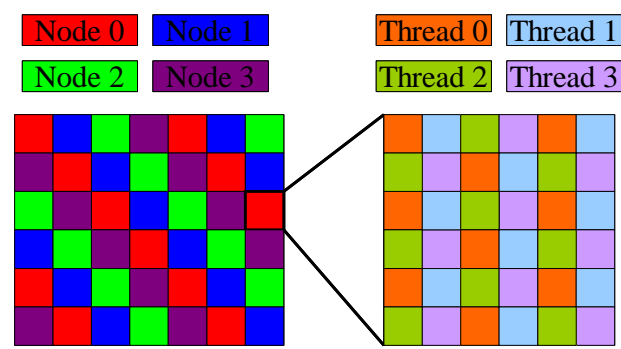


Figure 7: Partitioning of layout area among nodes and processors.

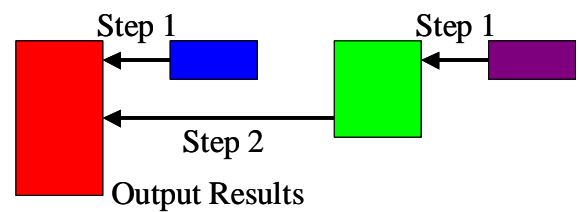


Figure 8: MPI Tree merge of results for four nodes.

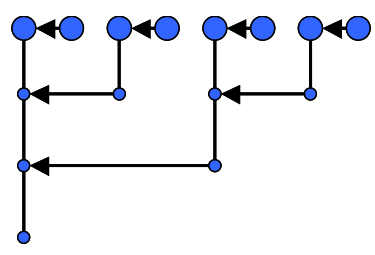


Figure 9: Tree merge for eight nodes with $\log_2 N$ steps.

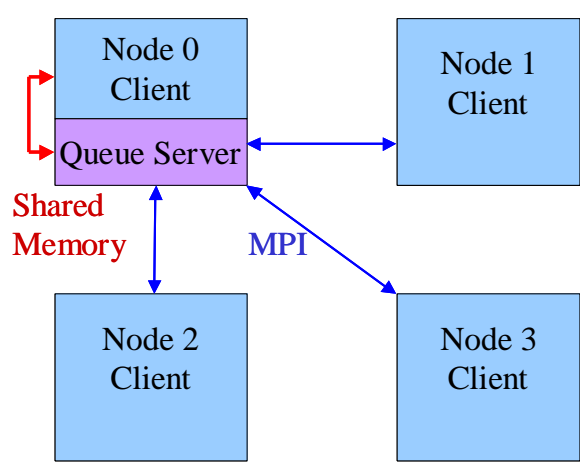


Figure 10: Dynamic load balancing queue client/server communication.

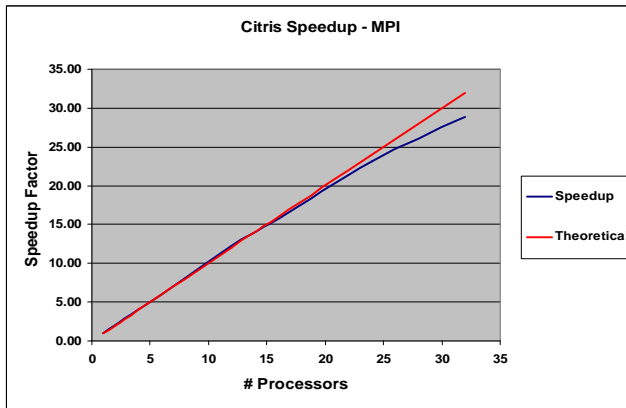


Figure 11: MPI speedup on Citris.

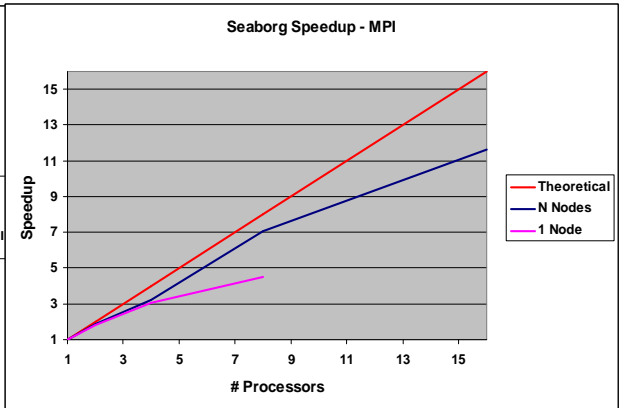


Figure 12: MPI inter-node /intra-node speedup on Seaborg.

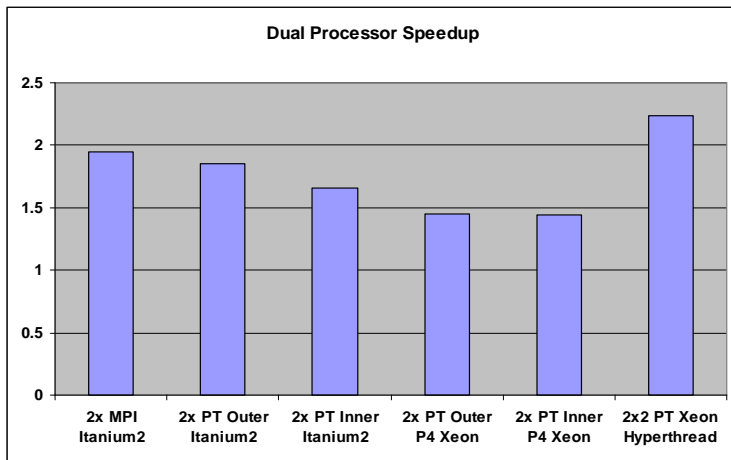


Figure 13: Dual processor speedup for various processor architectures.

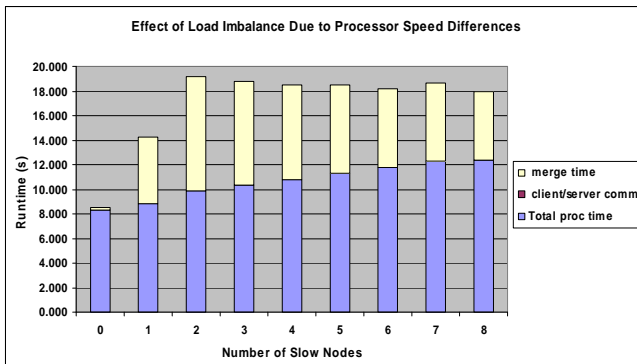


Figure 14: Breakdown of runtime with static load balancing.

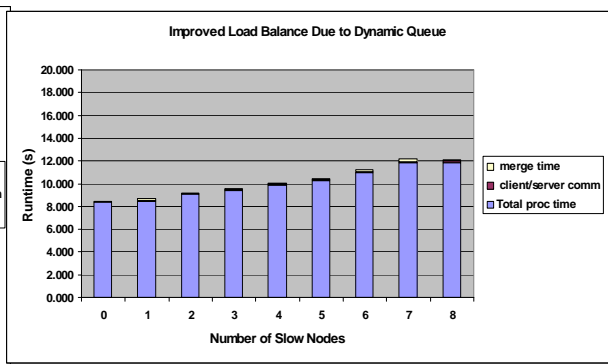


Figure 15: Breakdown and improved runtime with dynamic load balancing.