Distributed Sudoku Solver

December 14, 2011

Cha Li

Introduction

The motivation for this system is based on the fact that smartphones are extremely popular and provide a new computing platform that can be harnessed for distributed systems. Smartphones possess properties that make them more fit for a distributed system than there desktop and laptop counterparts. Firstly, smartphones are always connected to a data network. Secondly, smartphones are always on and available. Finally, smartphones have relatively high hardware specs which are rarely used at their full potential. The Distributed Sudoku Solver is an example of a use-case that harnesses the processing power of smartphones to solve a problem. On a larger scale, this system could be used to crowd-source search problems, number crunching, or simulations. The following will discuss the architecture, issues, implementation, and test results of the Distributed Sudoku Solver (also referred to as the "system").

System Infrastructure

Given the nature of this project, special hardware and software requirements had to be taken into consideration.

Hardware

The main hardware requirement imposed by this project is the use of smartphones (also referred to as "nodes"). Specifically, Android powered smartphones. Android was chosen over other platforms due to its open-source nature and ease of development. A dedicated machine, a laptop (referred to as the "server" or "central server"), would act as the central server with the main purpose of letting the operator interact with and manage the distributed system. The server is designed to be completely independent of the solving process. The final hardware requirement is a router which provided the communication infrastructure for the system. The communication is assumed to be reliable.

Software

Software development required two branches, server code and the Android application. The server code is mostly Java network code, listening and sending to sockets. Process scheduling is also the job of the server, and will be discussed later in the paper. The majority of development time went into the Android application since it has to include networking code, UI code, and Sudoku solving code. Application development used the Android SDK and is targeted towards Android 2.0.1 and above (API version 6+).

Design Issues

Before development began; the biggest issue was deciding the best way to split up a Sudoku board into smaller independent problems. Several options were considered such as splitting the board up by rows, columns, sectors (the bold outlined squares), numbers, or a combination of those. In the end, sectors were chosen as the smallest unit of division because a sector only needs to be compared to its four neighboring sectors whereas rows and columns need to be compared to all other rows and columns, respectively. Using sectors meant less internode communication.

During development, the biggest issue was implementing real-time scheduling into a Sudoku solver. The challenge came from the fact that Sudoku is inherently a non-real-time task. Squares within the Sudoku board aren't temporally dependent on other squares. The solution to this issue was having the server treat the nodes as independent processes which needed to be scheduled before they could run their task. A separate module was developed that could be simply "turned on" in the server which allowed for processes scheduling using pre-emptive EDF. Throughout the paper, the default scheduling method (non-EDF) will be referred to as "parallel" or "parallel scheduling.

Implementation

The system uses many aspects from the field of distributed systems which allow it to be scalable and robust. It can recover from nodes disconnecting for any reason, and it can intelligently manage nodes that join after it has started. In addition, since the server takes no part in the solving process it can die and the system will continue to solve the board (given no nodes die after the server dies). Load balancing is handled by the server and simply involves distributing, and redistributing, the sectors evenly among connected nodes. The implementation didn't stray that far from the initial proposal.

Architecture

The architecture of the system is somewhat of a hybrid. It's truly distributed when it comes to solving the Sudoku board (the main task). Each individual node keeps track of its own state and only communicates with nodes that contain neighboring sectors; as far as each node is concerned it's the only node solving something and others are there to help. When it comes to any type of system management (unrelated tasks to solving Sudoku), the server is needed. Tasks such as redistributing sectors for load balancing, handling new and dead nodes, and scheduling (when EDF is enabled) are handled by the server. All these tasks can be accomplished distributively, but the projects time-frame didn't allow for that.

Sudoku Solver

The Sudoku aspect of this project wasn't the main point of development; it was just a use-case to carry out the system idea. As such, the algorithms used to solve the Sudoku are pretty naïve and are limited to solving mostly "easy" and some "medium" difficulty boards. The biggest omission is the ability for nodes to make guesses. Guesses involve both forward-checking and maintaining arc-consistency which is an entirely separate project. This simplification also excludes the need for handling mutual exclusion. If the ability to guess was implemented, sectors (maybe entire nodes) would need to be locked while the guess was being verified. The following pseudocode outlines the loop each node iterates through until all its local sectors are solved:

```
initial remove of illegal values

while unsolved:
    for each local sector:
        for each square:
            if square has certain value
                fill

    pick a random sector
        pick an external neighbor of sector
        fetch state of external sector from external node

    if any external sectors have been received
        synchronize external sector with local sectors
```

Load Balancing, Fault Tolerance, and States

These two features are grouped together because given the nature of the system and the Sudoku game can be handled with the same mechanism: sector redistribution. The fault model of the system is N-1 nodes where N is the initial number of connected nodes. As long as one node remains connected, the system can continue solving the Sudoku board. A node can die for many reasons including: it's off, its battery is dead, it's not connected to a network, and/or the application is not running. In all these cases, the node is simply considered disconnected. Since all the parts of the Sudoku board are inherently dependent on all other parts of the board, a node dying means the sectors assigned to it will stop making progress. In turn, other sectors will stop making progress. Each node is hardcoded with a timeout value, T_n, which defines how long the node will continue while making no progress before it decides it's stuck. When the stuck decision is made, the node notifies the server of the situation and sends the state of its local sectors. From the server's point of view a stuck message doesn't only mean a node has died, it could also mean the system is legitimately stuck. Because of this ambiguity, the server also has a timeout value, T_s, which determines how long the server will wait for additional stuck messages after the initial stuck message. Once the timeout expires, the server takes the network addresses of the nodes that reported being stuck and redistributes the sectors among them. Sector redistribution is also used when new nodes join mid-solving. A new node is marked as "ready", not "running", and is queued for the next redistribution. One weakness of this is the system does not instantly perform load balancing; however, an advantage of this is that the currently running nodes will not be interrupted by the server every time a new node joins. The new node is only used when the running nodes need it.

One additional feature was implemented to take advantage of the sector redistribution mechanism, system throttling. Throttling allows the operator to purposely disable certain nodes or limit the performance of the system to some percentage of full capacity. In the scope of this project, this feature is mostly a toy but in real life applications a feature like this could be used to save power or isolate problematic nodes. Nodes that are throttled aren't disconnected, but put into the "idle" state. When the operator decides to make the nodes available again, the server puts them into the "ready" state queued for the next redistribution. The only situation where the server can override a user-throttled node is when the user throttles every node. In this case, the server forces one into the "ready" state.

Power Management

Originally, the plan was to use the Android SDK to create an application that could go into the "sleep" state on the smartphone; however, the time-constraints of the project dictated a simpler implementation. In its place a simple counter is used by the application to keep track of how many "power units" it uses. To make the measurements somewhat more realistic, several different values are used:

```
float    SEND_UPKEEP   = .5f;
```

The amount of power units it costs to send a message.

```
float    RECV_UPKEEP   = .7f;
```

The amount of power units it costs to receive a message. It's greater because the message also has to be processed.

```
float    ACTIVE_UPKEEP = 2.0f;
```

Used to calculate the cost of each iteration of the running loop while the node is active (solving).

```
float    IDLE_UPKEEP   = .3f;
```

Used to calculate the cost of each iteration of the running loop while the node is inactive (idle).

The following pseudocode outlines how the above values are used:

```
while running:
  if active:
    TOTAL_PWR += ACTIVE_UPKEEP * (# of local sectors)
  else:
    TOTAL_PWR += IDLE_UPKEEP * (# of local sectors)

function sendMessage
  TOTAL_PWR += SEND_UPKEEP

function messageProcessor
  TOTAL_PWR += RECV_UPKEEP
```

Real-Time Scheduling

As stated earlier, implementing real-time constraints into a Sudoku solver was a challenge. A solution didn't arise until very late into development. The scheduling feature was built as a module that could be enabled or disabled before the system started. The default option is no scheduler or the parallel "scheduler". Parallel scheduler just means all nodes are active all the time at the same time until the board is solved.

Real-time scheduling is done using EDF where each connected node is treated as a separate process by the scheduler. The implementation includes two parts, admission control and the actual scheduler. Under parallel scheduling all nodes that join are accepted and set to the ready state; however, under EDF all new nodes that join first have to go through admission control. Admission control does a few things; firstly it assigns a WCET and frequency value to the new node. These values are randomly generated from an accepted (predefined) range and are used to calculate the new nodes utilization value, U_i. Whether or not the node is accepted into the system is based on if it puts the system utilization value over 1.0 (100%):

```
if Ui + Us > 1.0
  reject
else
  accept
```

where U_i and U_s are defined as:

$$U_i = \frac{WCET_i}{freq_i} \qquad\qquad U_s = \sum_{i=1}^{N} U_i$$

Once a new node is accepted into the scheduler by admission scheduler, EDF takes over and decides where to add it in the queue of ready processes. EDF decides this by comparing the deadline of the new process (based on WCET, frequency, and other values) to the deadlines of current processes. The scheduler iterates through its loop once every millisecond "ticking" the current process. When the current process finishes or is pre-empted, it's stopped and reinserted into the queue. Then, the new head of the queue is marked as active and set to the current running process.

Since the scheduler is a global scheduler, it incurs one big performance hit: process serialization. Only one node will be active at any given time, even though the sectors of all nodes can be worked on independently. The graphs in the next section show some interesting side effects of the EDF scheduler on power usage and time required to solve the board.

Test Setup

The sizes of the tests were completely dependent on the number of smartphones I could get my hands on for this project. In this case I managed to get 5, so the tests were executed on systems with 1, 3, and 5 nodes. Testing was limited to a choice of one Sudoku board which acted as the control. The following is the board used in all experiments:

```
2 4 0 7 8 0 9 3 0
3 0 0 5 0 6 4 7 0
0 0 0 0 0 0 0 5 8
5 0 7 0 1 0 0 9 0
9 0 0 0 0 0 0 0 7
0 1 0 0 9 0 5 0 4
4 7 0 0 0 0 0 0 0
0 1 5 8 0 3 0 0 4
0 8 9 0 7 2 0 3 5
```

Experiments executed:
3x 1-Node EDF scheduler
3x 1-Node parallel scheduler
3x 3-Node EDF scheduler
3x 3-Node parallel scheduler
3x 5-Node EDF scheduler
3x 5-Node parallel scheduler

Notes:
All graphs that involve power measurements use the abstract "power unit" as the unit of measurement. When unlabeled, the X-Axis represents time.

Test Results: Power Usage

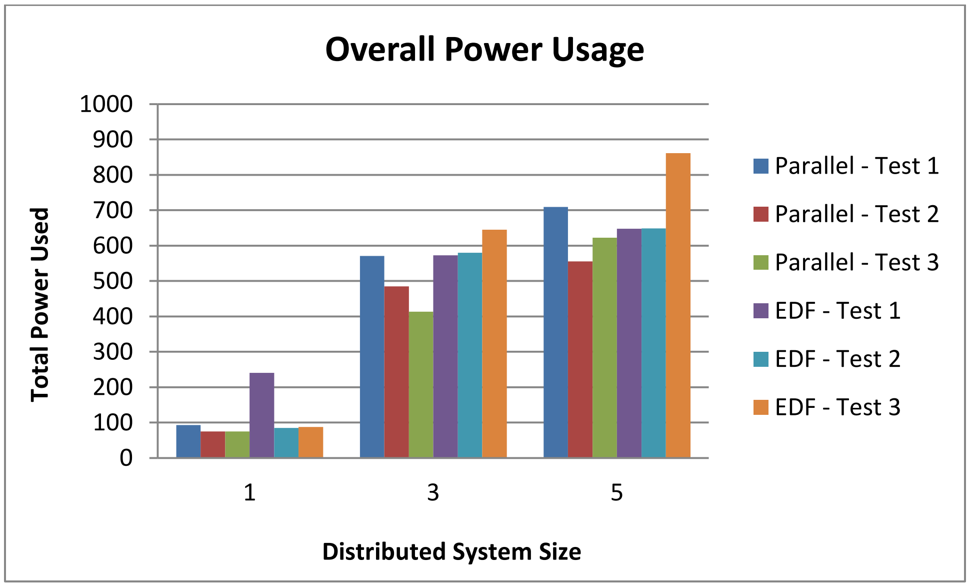

Figure 1: EDF and parallel scheduling use about the same amount of power when executing similar tasks.

The above graph doesn't show anything remarkable, aside from some outliers EDF and parallel scheduling use about the same amount of power in every test. In the following graphs, the power usage of the system is examined in greater detail and more interesting behaviors can be seen.

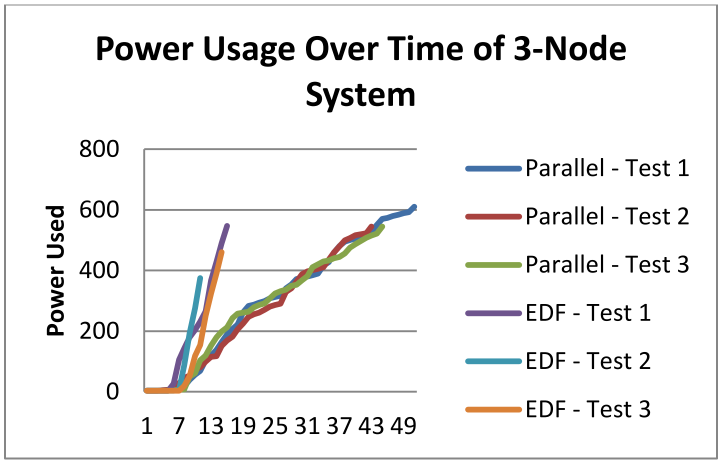

Figure 2

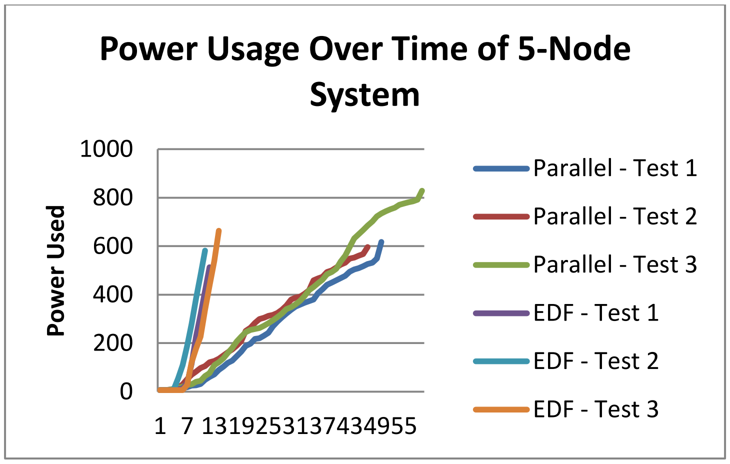

Figure 3

A couple interesting facts can be extracted from Figure 1 and Figure 2. Firstly, EDF scheduling takes much longer to solve a Sudoku board than parallel scheduling and secondly, EDF's power usage grows gradually over time whereas under parallel scheduling power usage shoots straight up. As stated previously, in the end they both use about the same amount of total power. The following two graphs take Test 2 from their respective scheduling methods and show the power growth of each individual node within that test (the above graphs are averages).

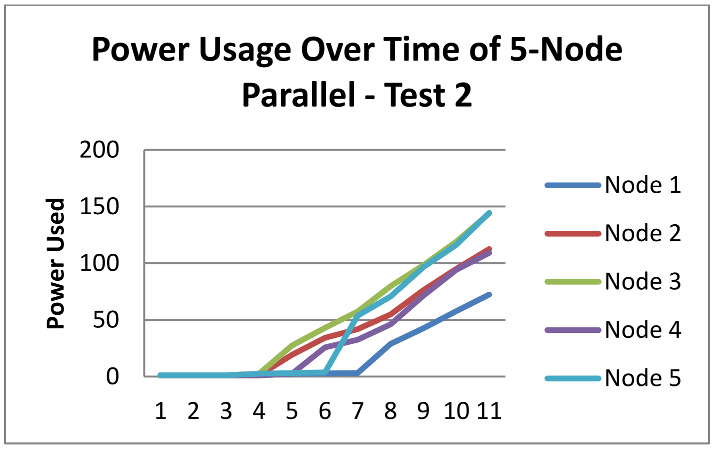

Figure 4

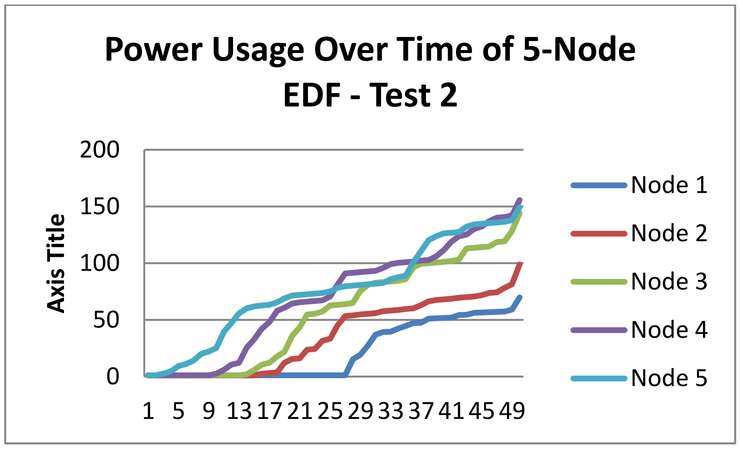

Figure 5

The initial flat-line for each node represents the time before the system starts, this part can be largely ignored. More importantly, are the slopes of the lines in the above two graphs. Not surprisingly, under parallel scheduling the lines are essentially linear since the nodes are always active. However, the slopes of the lines for EDF scheduling seem to oscillate between periods of high energy use and low energy use. This is an effect caused by the EDF scheduler. Under EDF, only one node is active within the system at a time and this is reflected in Figure 5 where, roughly, only one node experiences high energy usage within a given time period. It can also be said that the number of spikes in Figure 5 represent the number of times the node was scheduled to run by EDF. Perhaps the "squigglyness" of the lines in Figure 5 was caused by other nodes and overhead, in light of this the following graphs show power usage in a single-node system under each scheduler.

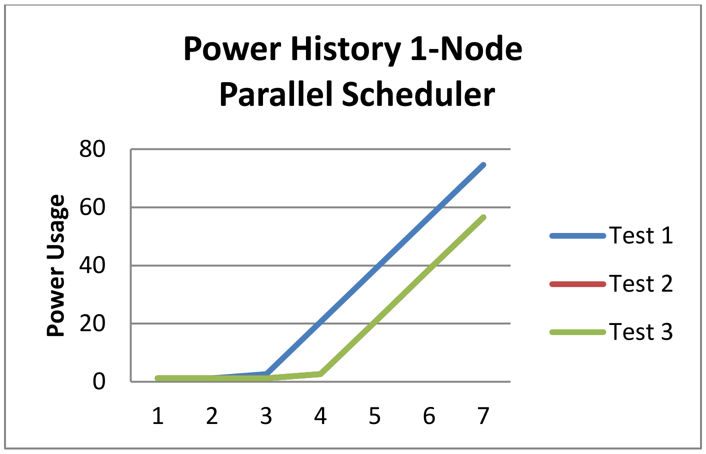

Figure 6

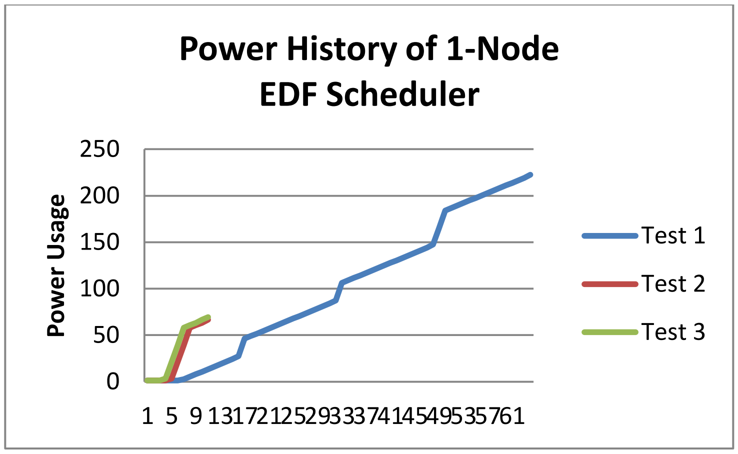

Figure 7

It’s pretty safe to say that EDF has a direct influence on how much power the system uses over time.

Test Results: Scheduling

Some of the effects of the EDF scheduler have already been seen in the graphs analyzing power usage. The following graphs show how EDF actually handles the processes.

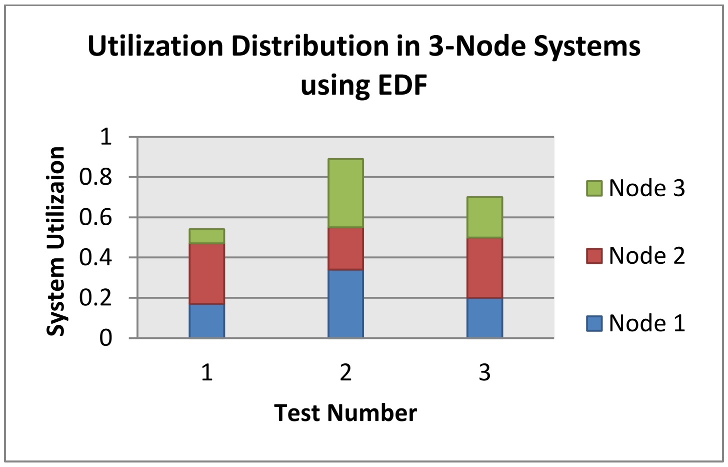

Figure 8

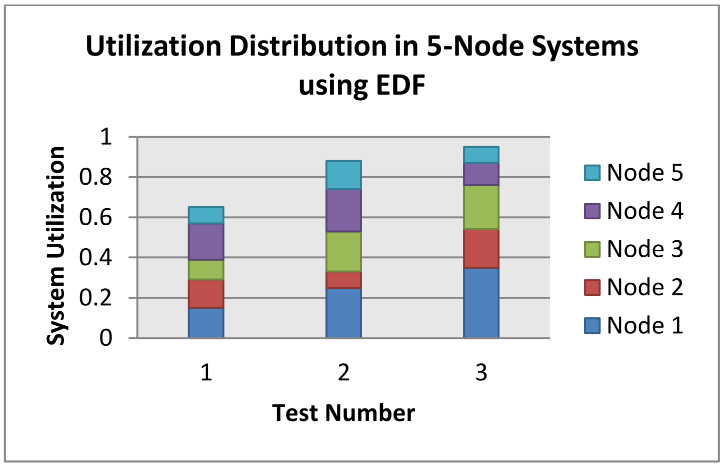

Figure 9

Figure 8 and Figure 9 simply visualize the equation for calculating U_s. Since EDF is guaranteed to not miss deadlines if $U_s < 0$, I did not include any statistics on missed deadlines (admission control does now allow utilization to go above 1.0).

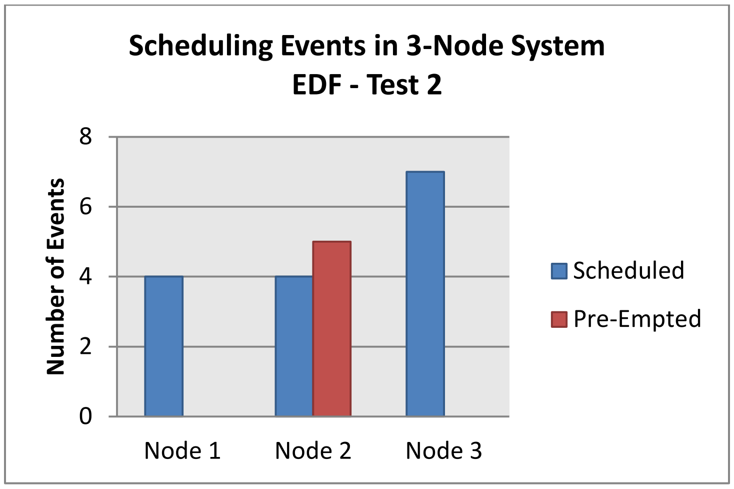

Figure 10

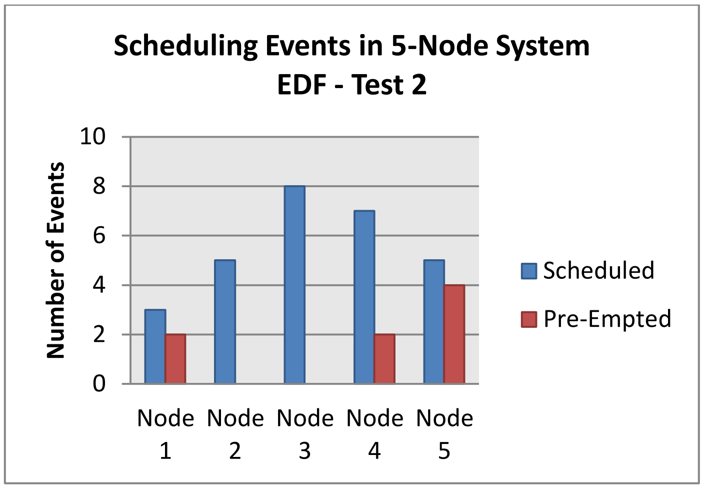

Figure 11

The interesting thing here is comparing the number of "Scheduled" events for each node in Figure 11 and comparing that number to the number of spikes for each node in Figure 5. However, it should be noted that the history data gathered by the system does not include the last spike due to how nodes end running. This means the number of "scheduled" events will be one less than number of spikes.

Additional Observations

One of the other things I noticed, which isn't clearly represented by the above graphs, is that the lower the system utilization is the more power is wasted by the system through idling. With parallel scheduling, all tasks run simultaneously, constantly active until the puzzle is solved. In some cases a node might idle for a small amount of time if it's waiting for new sector information. Under EDF, however, it's possible a lot of time is spent idling due to the serialization of nodes. If EDF always ran at near 100% utilization, very little power would be wasted. This fact is most evident in the line representing "Test 1" in Figure 7.

Conclusion

I'm finally done.