HPC Challenge report

Submitted by:	Benny Raphael
Department:	Building
NUSNET ID:	bdgbr
Category:	Serial to parallel
Systems used:	NUS HPC Linux Clusters
Software used:	Custom code written in C using MPI library. The newly developed C program calls an existing freeware lighting simulation software *radiance*. Radiance 3.7 was recompiled to run on the atlas4 cluster
Project title:	Computation of lighting energy consumption using parallelized day lighting simulation
Project members	Benny Raphael

Computation of lighting energy consumption using parallelized day lighting simulation

Dr. Benny Raphael

1. Problem definition

Lighting control systems are gaining increasing attention due to rising energy costs and enhanced environmental consciousness. Advanced lighting control systems involve the use of dimmable ballasts and light sensors and are capable of automatically dimming the lamps when adequate daylight is available. Such systems have the potential to save significant amount of energy when buildings are designed with good day lighting features. Naturally, building designers are interested in computing the savings in the energy consumption of artificial lights for evaluating their designs.

In order to compute the energy consumption of electrical lights, lighting simulations have to be performed. Radiance (Ward and Shakespeare, 1995) is a lighting simulation software that employs backward ray-tracing. This software is being used as the underlying simulation engine in many popular building design software and has been extensively tested and validated.

Computation of lighting energy consumption using the simulation software Radiance involves the following steps:
Step 1: Model creation: Building geometry, material properties and weather data are used to create a simulation model which is compiled into an octree representation. The position of the sun (depends on the hour of the day that is simulated) and the sky parameters are part of this model.
Step 2: The illuminance at various points in the interior of the building is evaluated using the octree model. This is typically done along a grid of spacing 1 meter or less. An important point to note here is that ***the illuminance at any point can be evaluated independently of other points***. In other words, this part of the process is naturally parallel. The problem is very similar to the rendering of a pixel in 3D computer graphics. However, in most commercially available lighting simulation packages the computation of illuminance of points is done sequentially. In Radiance, a tool called *rtrace* reads points one by one and outputs the illuminance values sequentially.
Step 3: The average illuminance value that is measured by a light sensor is predicted by taking the average of all the points within the area influenced by the sensor.
Step 4: Using the predicted average illuminance value, the level to which the lights can be dimmed are calculated. The power consumption of the lamp is calculated by assuming linear relationship between dim level and the power.
Step 5: The total power consumption of all the lamps at the particular hour is calculated.
Step 6: Steps 1 to 5 are repeated for different hours of the day (simulation hours) in order to obtain hourly power consumption. The total energy consumption is calculated by summing up the energy for each simulation hour.

The above steps are summarized in the form of a flow chart in Figure 1. There are two nested loops. The outer loop repeats for all the hours of simulation, typically from 8 am to 7 pm when day light is available. The inner loop repeats for each grid point where the illuminance level is to be computed. (Other loops involving computation of sums and averages are not shown in the flow chart because they do not take a lot of CPU time). The part that takes most amount of time is the computation of illuminance of grid points. For a small problem of average complexity, the whole process can take up to 8 hours to complete. This project aims to reduce the execution time through parallel computing.

1.1 Test case: BCA Zero Energy Building

In order to illustrate the reduction in execution time through parallelization, an example of a real building is taken in this study. This is the newly constructed Zero Energy Building of the Building Construction Authority (BCA) of Singapore, located at 200 Braddell Road. The second floor library of the building is used for lighting simulation and energy computation. This zone has an area of 18 m x 36 m. The whole building is included in the simulation model, however, the illuminance is computed only for the second floor library zone. A grid of spacing 1 m x 1m is used which contains a total of 648 grid points.

Trees on the west side of the building are also included in the model. This results in thousands of reflective planar surfaces and increases the simulation time considerably. A rendering of the building model generated by Radiance is shown in Figure 2.

1.2 Significance and impact of research

Lighting simulation and energy computation are very practical tasks and are becoming increasingly important due to the current efforts towards mitigating the effects of global warming and climate change. At present, building design options are not systematically evaluated because of the difficulties involved in the modeling and simulation. The time taken for completing the simulations is also a significant barrier to the wide spread adoption of simulation tools. Reducing the computation time through this research is expected to remove this barrier.

1.3 Problem size and complexity

Lighting simulation is a combinatorial problem having exponential complexity with respect to the number of surfaces involved in the model. This is because the illuminance of any point depends on the direct and diffused components of reflections from all combinations of surfaces. Since the selected example contains thousands of reflective surfaces the number of combinations is enormous. In practical simulations, the complexity is managed by specifying parameters such as the ambient bounce which restrict the number of reflections that are considered in evaluating a point. Even with this parameter set to as low as 6, the execution time is significant.

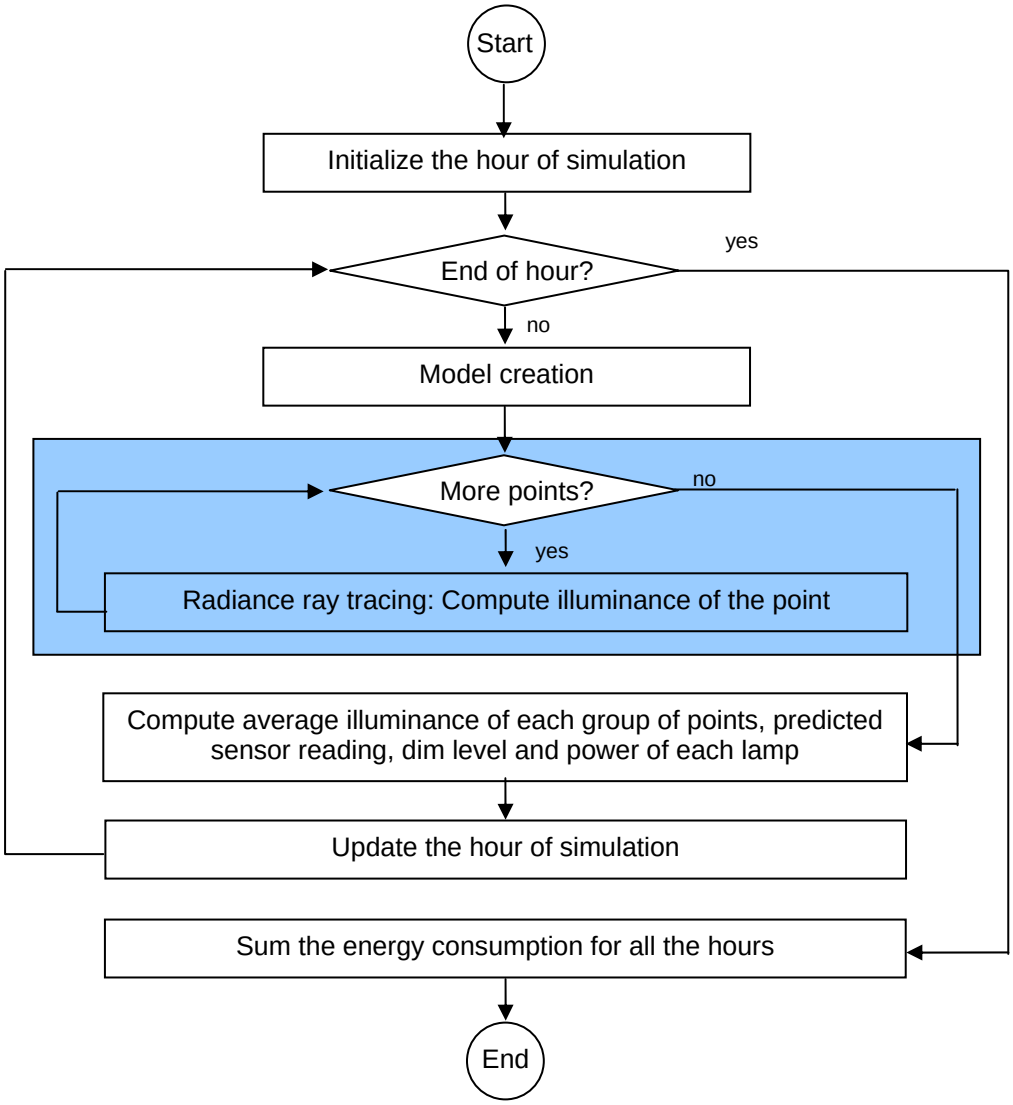


Figure 1. Conventional serial algorithm for computing the energy consumption of artificial lights. The box shown in blue is a loop that is naturally parallel. The illuminance of each point can be computed independently of others.

Once the simulation parameters such as the ambient bounce are fixed, the execution time varies linearly with the number of hours simulated (H) and the number of grid points (G). The execution time can be calculated approximately using the following formula.

$$T(H, P) = O_1 + H \times (O_2 + G \times t_p)$$

Where

H is the number of hours simulated

G is the number of grid points

O_1, O_2 are the overhead time constants

t_p is the time taken for computing the illuminance of a single point

O_1 includes the time taken to initialize the process, read input data etc. O_2 includes the time required to compile the simulation model for the hour. A significant part of this involves manipulating model files.

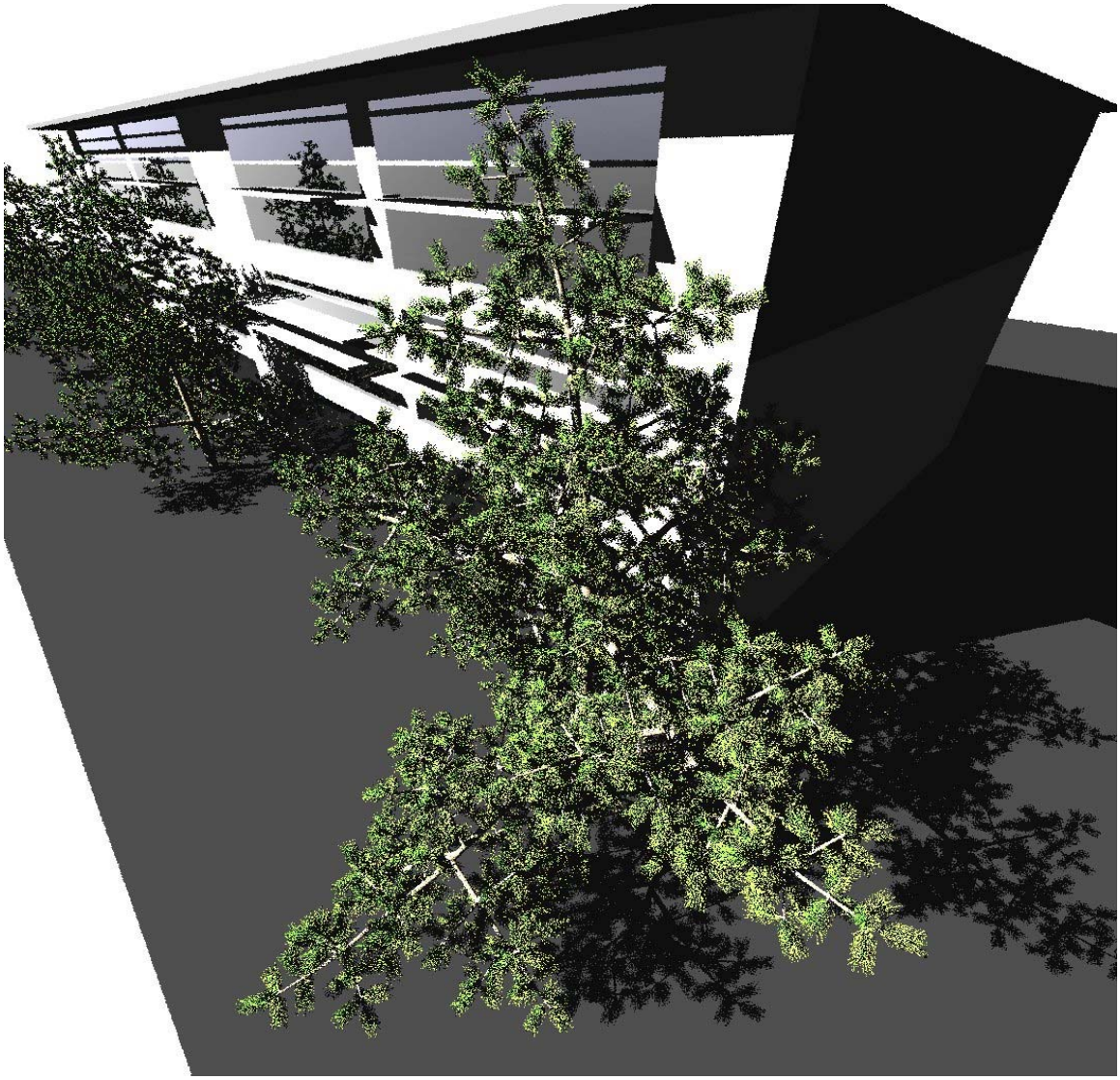


Figure 2. Rendering of the model of the BCA-Zero Energy Building at 2 pm generated by the Radiance simulation software. Models of trees are taken from standard libraries and do not correspond to the real ones at the building site.

2. Solution Strategy

The key to parallelization is the separation of the algorithm into parts that can be computed independently. Here, the computation of the illuminance of grid points can be computed independently after the model has been compiled for a particular simulation hour. In the first attempt to parallelization in this project, the grid points were divided equally among all the parallel processes for each hour. However, this did not give good results because of too many synchronization points involved. **Every process had to wait for the compilation of the model for each hour**.

The final solution involved small changes to the serial algorithm shown in Figure 1. First of all, the model creation part is taken out of the main loop spanning the entire algorithm. Instead the models for all the simulation hours are created upfront. This ensured that processes need not wait for the completion of model creation after each simulation hour. Since the models are created a-priori, processes could proceed to the next simulation hour immediately after finishing one.

Secondly, the problem was decomposed into chunks of approximately equal size so that the load on every process was balanced. Each problem chunk consists of a set of grid points for the same simulation hour. One simulation hour may be divided into one or more problem chunks depending on the load balancing requirements. Each process solves one or more problem chunks. This is explained in more detail in Section 2.1.

The final parallel algorithm is shown in Figure 3. There are two synchronization points. In the first part, simulation models are created for each simulation hour. Only the first process compiles the simulation model. The output which is a compiled octree model is saved in the shared folder accessible to all the processes. This part of the program cannot be executed in parallel because the model files need to be read and manipulated.

The second loop shown in the blue box is the most important one and takes most of the computation time. In this loop, each process uses the compiled model for the simulation hour to compute the illuminance of grid points within each problem chunk allocated to it.

After a process has completed a problem chunk, it sends a notification to the first process. After finishing the problem chunks allocated to it, the first process waits for receiving notification from all the processes. After this, the first process performs the remaining steps. The remaining steps involve simple arithmetic operations such as summing and averaging and are computed within a couple of seconds. Implementation of the main routine of the algorithm in the form of C code is given in Appendix 2.

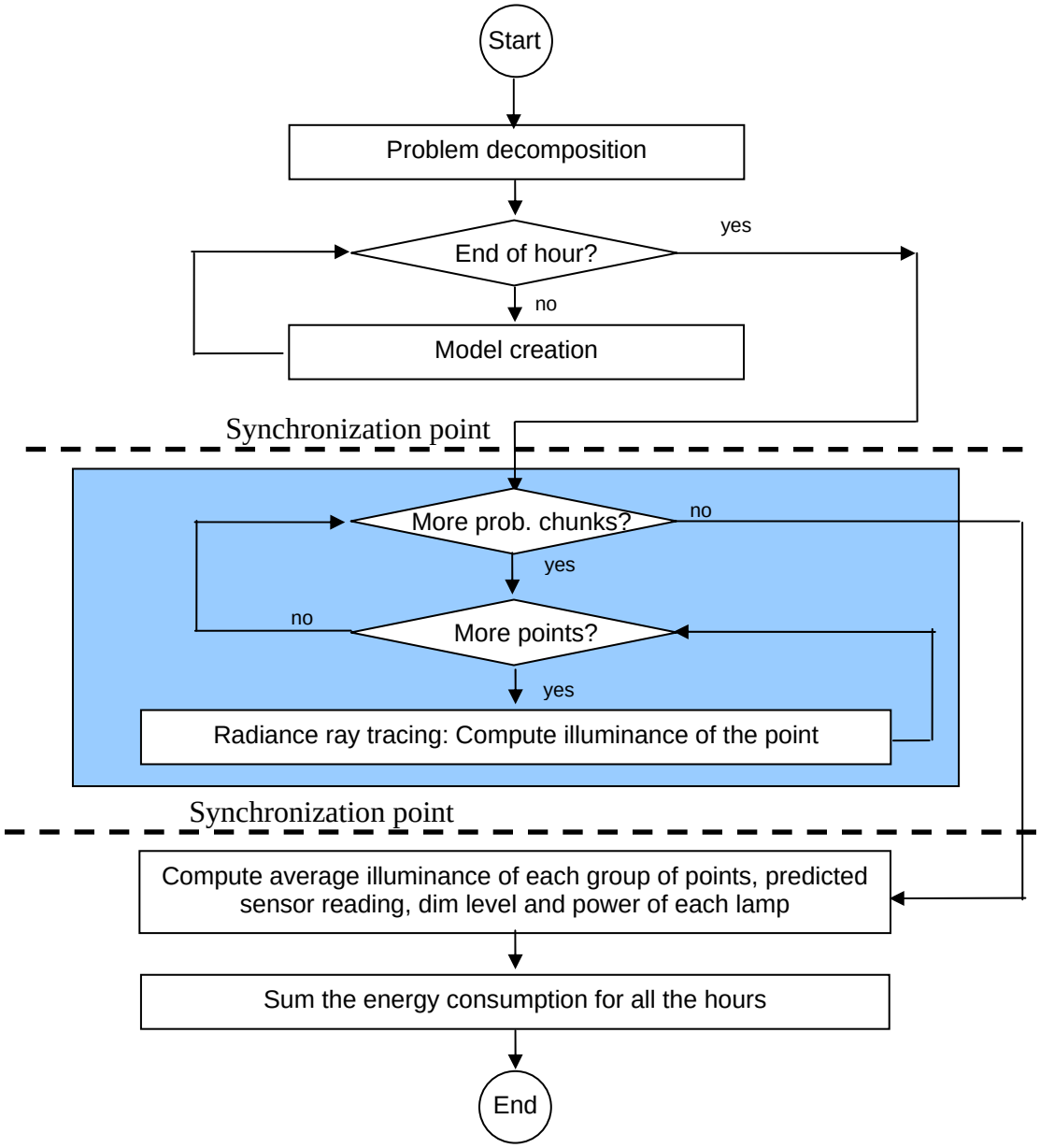


Figure 3. Parallel algorithm for computing the energy consumption of artificial lights. The box shown in blue is the loop that is executed in parallel. The problem is decomposed into chunks such that the load on every process is approximately equal.

2.1 Problem decomposition algorithm

The objective of problem decomposition is to distribute the load among multiple processes as uniformly as possible. Since synchronization points become bottlenecks, it is necessary to divide the problem such that minimum communication is needed among the processes cooperating in solving a part of the problem. In this application, processes that compute the illuminance of grid points for the same simulation hour need to cooperate for sharing the compiled model. Therefore, the grid points for each simulation hour should be divided among minimum number of processes. For example, if there are 12 simulation hours and 6 processes, the execution time is minimum if each process solves exactly two simulation hours. In this case, there is no need for two processes to share the same model. However, if there are 6 simulation hours and 12 processes, in order to balance the load among all the processes, each simulation hour has to be divided equally among two processes. The decomposition algorithm should also handle cases where the number of hours and processes are not exact multiples of each other.

The decomposition algorithm developed in this project is described below:
Step 1: Distribute maximum number of full simulation hours to each process. Calculate the number of full hours per process (nhp) by the integer division of number of hours (H) by the number of processes (P).

$$nhp = H / P \qquad \text{--- (Eq 2.1.1)}$$

Calculate the remaining number of hours (rem) by taking the remainder of this division. The number of big problem chunks is equal to the product of number of full hours (nhp) and the number of processes (P)

Step 2: Divide the remaining hours (rem) among all the processes. Calculate the number of grid points in the smaller problem chunks (gs) by dividing the total number of grid points remaining for all the hours (G x rem) by the number of processes (P).

$$gs = G \times rem / P \qquad \text{--- (Eq. 2.1.2)}$$

Step 3: Repeat for each process: Create nhp Problem chunks for the process. Starting with the first simulation hour, allocate nhp number of hours to each process sequentially.

Step 4: Initialise the current process curp as 0 (first process has index 0). Repeat for each remaining simulation hour: Create a Problem Chunk of size gs for the current process, Increment curp until the grid points for the hour are completed. The last problem chunk for the hour might have less than gs number of points, in which case, curp is not incremented and the first problem chunk of the next simulation hour is allocated to the same process. In this situation the next problem chunk will have less than gs points so that the total number of grid points allocated to curp is kept equal to gs.

C language code implementing this algorithm is given in Appendix 1. As an illustration, the simulation hours and grid points allocated to 16 processes when there are 12 simulation hours and 648 grid points is shown in Table 1.

Problem chunk	Simulation hour	Process allocated	Number of grid points in the problem chunk
1	8	1	486
2	8	2	162
3	9	2	324
4	9	3	324
5	10	3	162
6	10	4	486
7	11	5	486
8	11	6	162
9	12	6	324
10	12	7	324
11	13	7	162
12	13	8	486
13	14	9	486
14	14	10	162
15	15	10	324
16	15	11	324
17	16	11	162
18	16	12	486
19	17	13	486
20	17	14	162
21	18	14	324
22	18	15	324
23	19	15	162
24	19	16	486

Table 1. Decomposition of a problem consisting of 12 simulation hours and 648 grid points among 16 processes. Each problem chunk consists of a set of grid points for an hour. Each process solves one or more problem chunks.

2.2 Synchronization issues

Since the first process performs the compilation of models for every hour, remaining processes have to wait until their first model is ready. Model compilation has to be done sequentially because model files need to be manipulated and parallel access to the same files are not advisable. In order to reduce the waiting time of remaining processes, models are compiled in a specific order. The models for the first simulation hour of each process are compiled first. As soon as a model has been compiled, all the processes that use the model are sent notifications so that they can start immediately while the first process is compiling remaining models.

Notifications are implemented using two methods, using MPI functions and using file locks. Either method could be chosen by setting a flag. It is found in some cases that even after a process receives a notification that a model is ready, the file containing the

compiled octree is not immediately readable. Therefore, an additional check has been implemented to make the process wait until the file can be opened successfully. It is assumed that these problems are due to the simultaneous access of shared network folders. The code for implementing notification using MPI is given in Appendix 3.

2.3 Analysis of the execution time in the parallel implementation

Model compilation takes about 15 seconds for each simulation hour. If there are 12 simulation hours, this results in 180 seconds of additional computations for the first process. This is insignificant compared to the total execution time which is in the order of thousands of seconds. It is however admitted that as the execution time reduces through the use of more processes, the compilation time could make a non-negligible effect.

In practice, it has been found that the extra work done by the first process does not have any effect on the final execution time because some other process always completes last. This is due to the variations in the load on different machines in the cluster and variable communication delays in accessing shared folders. Empirical tests conducted on the cluster indicate that File I/O takes a significant part of the execution time.

If you assume that every process has equal probability of being scheduled on a machine with a high load (either processor or network), the probability of the first process taking more time than others is only (1/P) where P is the number of processes. The probability of any other process getting scheduled on a highly loaded machine is (P-1)/P, that is, it is (P-1) times higher. Therefore, the additional 180 seconds spent by the first process has negligible effect on the total execution time. This is illustrated in Figure 4. In this illustration, the time for each simulation is assumed to vary between 400 and 600 seconds (in actual run, each simulation takes about 2000 seconds and there is wider variability). Process 2 is assumed to be scheduled on a highly loaded machine compared to the rest. In this case, the first process has to wait until Process 2 completes all the simulation.

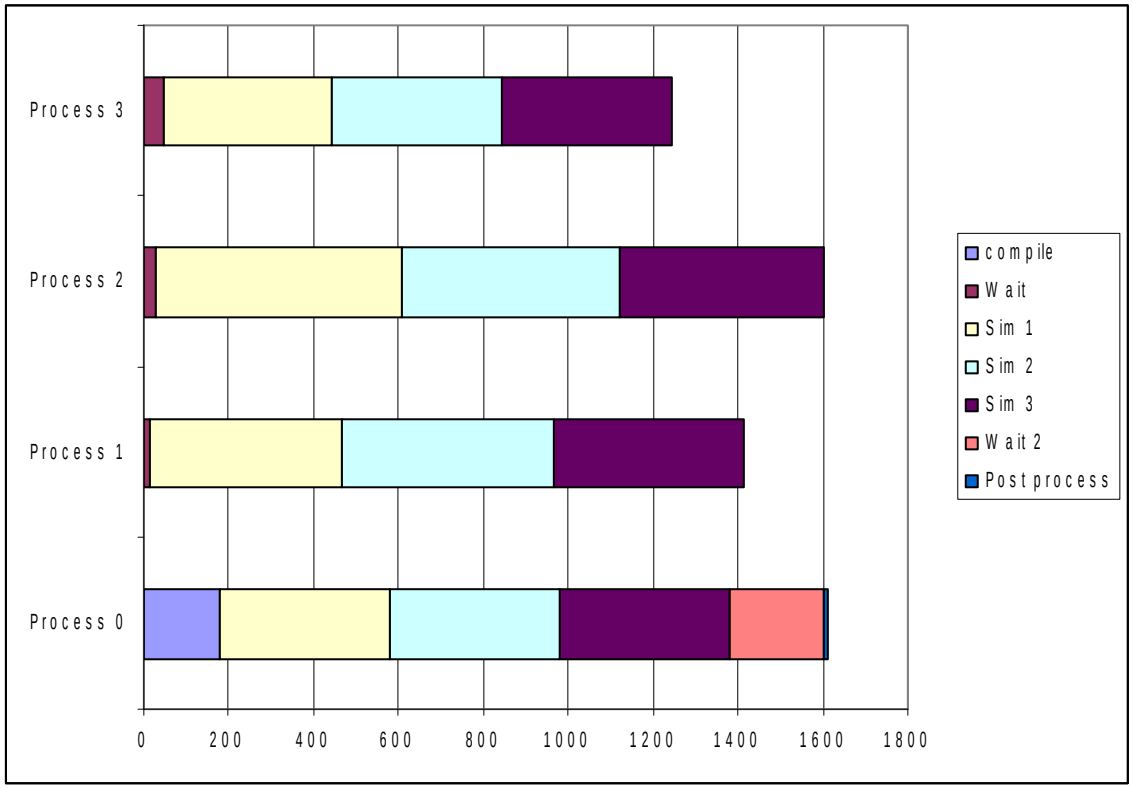


Figure 4. Illustration of parallel scheduling of tasks. Even though, the first process performs the additional task of compiling the models for all the simulation hours and starts last, it has been found in all the runs that it has to wait for other processes to complete all the simulations.

3. Results

Final energy consumption obtained through the linux cluster implementation are very similar to those obtained using a serial implementation on Windows. The minor differences are due to some variations in default parameters in the version of Radiance used on the Linux platform. A plot of the brightness values obtained for the simulation hour 14:00 is shown in Figure 5. The lighting power consumption for different simulation hours are plotted in Figure 5.

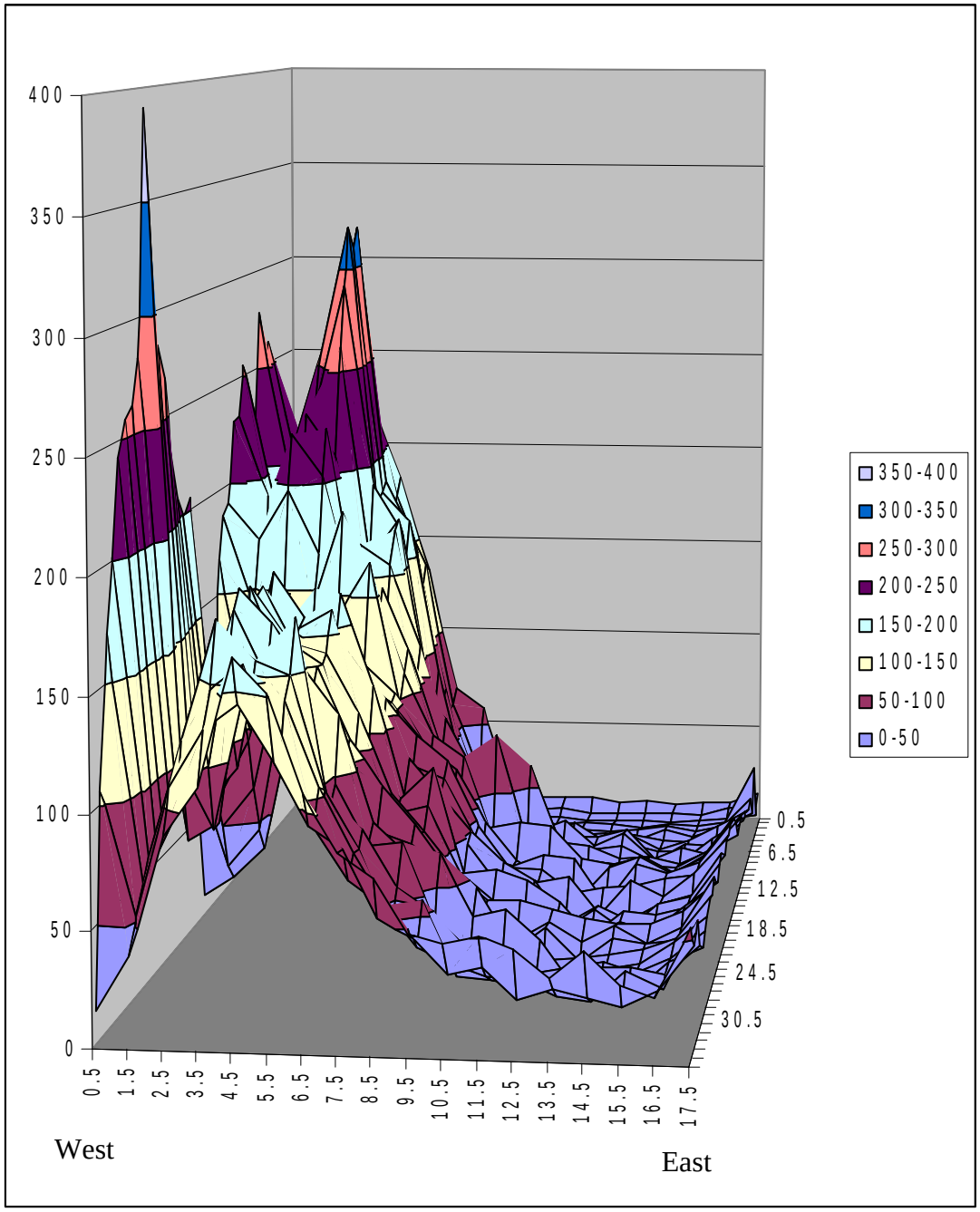


Figure 5. Surface plot of brightness (lux) at every grid point at 14:00 hours in the BCA Zero Energy Building. Points on the west side have higher illuminance because the sun is on the west side at this time.

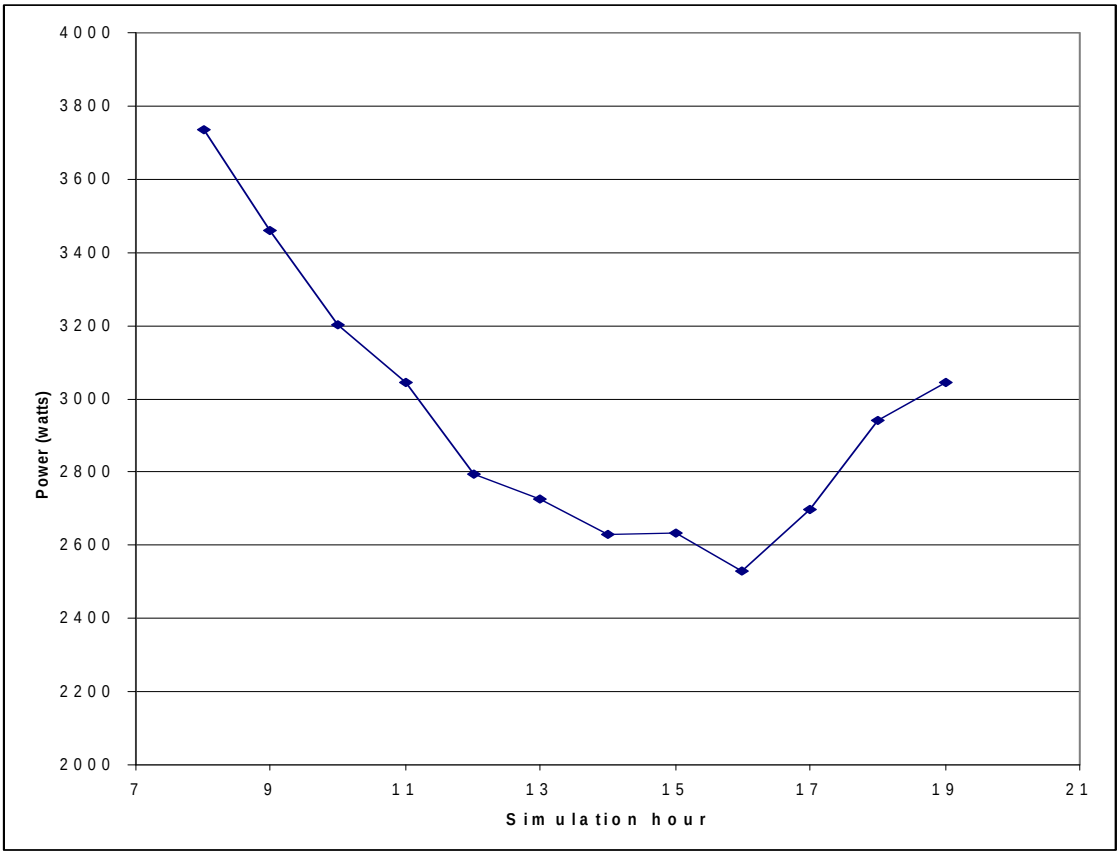


Figure 6. Predicted lighting power consumption for all the simulation hours. Power consumption is minimum late in the afternoon when the sun is on the west because of the daylighting features on the west side of the building.

3.1 Speed up

In order to evaluate the speed up obtained, the problem was solved using different number of processes starting from 4 up to 16. In order to calculate the time required for the serial execution the following procedure was used because the MPI job submission script does not accept less than 4 processes: A parameter was read from a setup file which specifies the actual number of processes to be used. If this parameter is 1, then the problem is decomposed into only one process. That is, all the grid points and simulation hours are assigned to the first process. Other processes terminate without performing any computations. Thus the problem is solved serially using a single process and the execution time is recorded.

The above series of executions are repeated three times on the atlas4 cluster (using the queue quad_parallel). After the new cluster nodes based on Intel Nehalem 2.66GHz quadcore processors were added, the executions were repeated three times on these nodes (using the queue hpc2009). The best of six runs is taken for the evaluation of the speed up

factor. The best value is taken instead of the average because of the wide variation in the execution times from one run to the other. The actual execution time depends largely on the existing load on the machine that is allocated to each process. Therefore, in these runs the objective is to get an estimate of the best execution time that is possible with a given number of processes. It is noted that in general, there is lesser variability in the runs on hpc2009 queue and the time taken is lesser, because the load on the machines were very low during the initial days after the new nodes were added.

The speed up factor for P processes is defined as the ratio of the execution time for 1 process (serial process) and that for P processes.

$$S(P) = T(1)/T(P)$$

Where T(P) is the execution time for P processes. The efficiency of speed up is the ratio of speed up and the number of processes. An ideal parallel process has an efficiency of 1, that is, the speed up is directly proportional to the number of processes.

The execution times and speed up parameters are summarized in Table 2. The columns Altas4 1 to Atlas4 3 give the execution time in seconds on the old cluster nodes. The columns HPC 1 to 3 contain the execution time on the new cluster nodes.

Num Processes	Atlas4 1	Atlas4 2	Atlas4 3	HPC 1	HPC 2	HPC 3	Best time	Speed up	Efficiency
1	28231	22195	22624	20601	19639	19531	19531		
4	7173	8260	7093	5137	5052	5090	5052	3.87	0.97
5	5846	6593	6877	4747	4846	4836	4747	4.11	0.82
6	6116	4901	4318	3563	3488	3606	3488	5.60	0.93
7	4502	5941	4035	3716	3644	3678	3644	5.36	0.77
8	3684	6044	3370	3051	3088	3019	3019	6.47	0.81
9	3713	4394	3307	2867	2867	2756	2756	7.09	0.79
10	3246	4395	3457	2835	2733	2879	2733	7.15	0.71
11	3347	4068	2837	2719	2672	2800	2672	7.31	0.66
12	2664	3192	2583	1804	1851	2013	1804	10.82	0.90
13	2330	2242	2073	2454	2537	2560	2073	9.42	0.72
14	2831	2571	2432	2463	2313	2604	2313	8.44	0.60
15	2607	2337	2155	2402	2229	2591	2155	9.06	0.60
16	2180	3933	2218	2320	2084	2279	2084	9.37	0.59

Table 2. Summary of execution times and speed up.

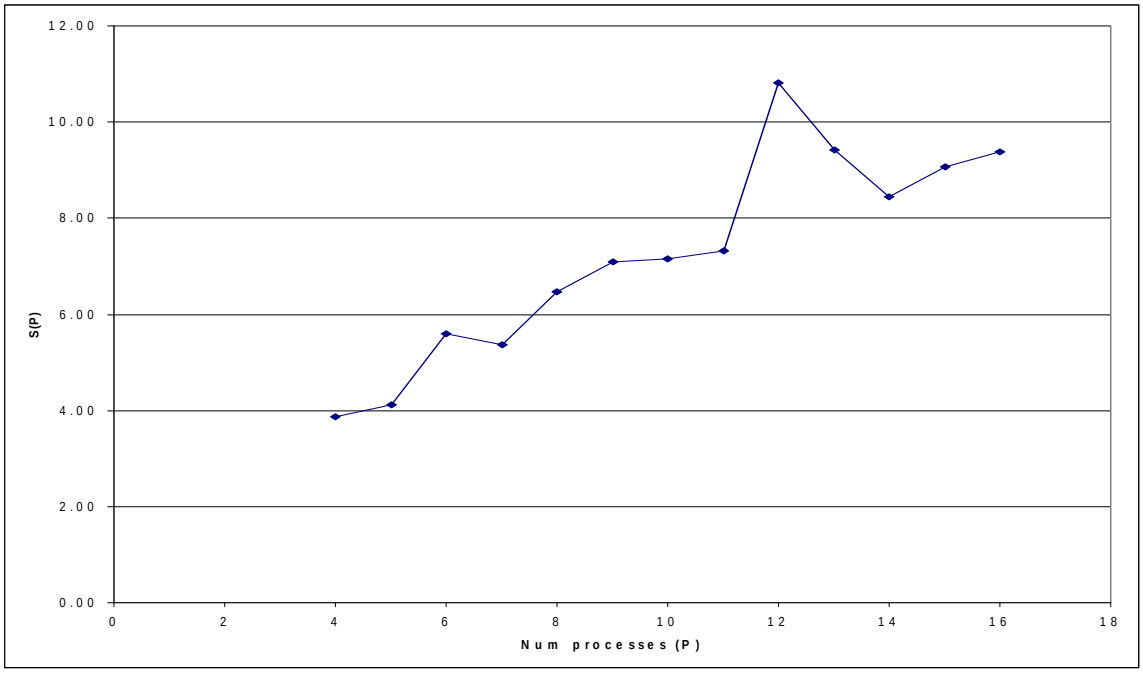


Figure 7. Speed up as a function of the number of processes. Speed up as high as 10.8 has been achieved.

The maximum speed up obtained was 10.8 using 12 processes. That is, it is possible to reduce the execution time by about 90%. **The problem which took about 8 hours to complete on a windows PC using a serial implementation was able to complete within less than an hour on the parallel cluster**. This is a remarkable achievement.

Efficiency as high as 0.97 has been obtained. The efficiency tends to drop as the number of processes is increased. This is partly related to the load on the cluster and the network communication delays. As explained earlier, when the number of processes increases, the probability that one of the processes is scheduled on a heavily loaded machine (or a machine with high network traffic) increases. The final execution time depends on when this process terminates. Multiple runs using the same number of processes show a variation as high as 1700 seconds in the execution time. This is a result of just one process getting delayed due to the existing load on that specific machine.

4. Concluding remarks

Significant speed up has been achieved by parallelizing the computation of lighting energy consumption. An existing problem that took about 8 hours to complete on a windows machine using a serial alogorithm could be completed in less than an hour on the parallel computing cluster. The maximum speed up obtained was 10.8 using 12 processes and the average efficiency was 0.76. This shows that the problem is linearly scalable.

The code for parallelization of lighting simulation that was developed in this project is very generic and can be used to perform the simulation of any building. Users need to input only the building geometry, properties, weather data and simulation hours. The program automatically decomposes the problem and executes it in parallel on available processes. This is a valuable tool for engineers and building designers.

References

Ward Larson G, Shakespeare R. (1998). Rendering with Radiance: the art and science of lighting visualization, San Francisco: Morgan Kaufmann.

Appendix 1: Code for problem decomposition

```
/* Structure to represent a problem chunk. Each problem chunk contains a set of grid points for a simulation hour.  */
struct ProblemChunk {
       int hour;                          /*  The simulation hour  */
       int startPoint;                    /*  The index of the starting grid point for this chunk   */
       int endPoint;                /*  The index of the ending grid point for this chunk   */
       int process;                 /*  The process which solves the problem chunk  */
};

/* Decomposes the problem into smaller chunks that can be executed in parallel.
       Uses the following global variables
              startTime       - starting simulation hour
              endTime         - ending simulation hour
              numProcess      - Number of parallel processes used
              numGrid         - Number of grid points
*/
int decomposeProblem() {

       int numHours;
       int sizeSmallChunks;
       int i, curp, remain, curgrid , hour, curchunk, allocated;
       int numHoursPerProcess, numProcessPerHour, remainingHours, numSmallChunks;

       numHours = endTime - startTime + 1;

       numHoursPerProcess = numHours/numProcess;    /* Number of full hours per process */
       remainingHours = numHours % numProcess;     /* Remaining hours that cannot be assigned completely to a single process */
       if (remainingHours > 0) {
              numProcessPerHour = numProcess/remainingHours + 1;
              if (numProcess % remainingHours != 0) numProcessPerHour ++;
       } else numProcessPerHour = 1;               /* To avoid division by zero */

       sizeSmallChunks = remainingHours * numGrid/numProcess;      /* Size of smaller chunks  */
       numSmallChunks = remainingHours * numProcessPerHour;    /*  Maximum number of small chunks */
       if (remainingHours == 0) numSmallChunks = 0;            /* if the num hours is an exact multiple of the number of hours */
       numProblemChunks = numProcess*numHoursPerProcess + numSmallChunks + 1;

       problemSizeForProcess = (int *)calloc(numProcess, sizeof(int) );
       pchunks = (struct ProblemChunk *)calloc(numProblemChunks, sizeof(struct ProblemChunk) );
       if (pchunks == NULL) {
              printf("Memory not available\n");
              exit (1);
```

```
        }

        /* Initialise */
        for (i=0; i< numProblemChunks; i++) pchunks[i].process = -1;
        for (i=0; i < numProcess; i++) problemSizeForProcess[i] = 0;   /* Total Number of grid points allocated to a process in the
smaller chunks */

        hour = startTime;
        curchunk = 0;

        /*  Allocate big chunks */
        for (curp=0; curp < numProcess; curp++) {
                for (i=0; i < numHoursPerProcess; i++) {
                        pchunks[curchunk].hour = hour;
                        pchunks[curchunk].process = curp;
                        pchunks[curchunk].startPoint = 0;
                        pchunks[curchunk].endPoint = numGrid -1;
                        allocated = pchunks[curchunk].endPoint  + 1 - pchunks[curchunk].startPoint;
                        curchunk++;
                        hour ++;
                }
        }

        /* Allocate small chunks  */
        curgrid = 0;
        curp=0;
        while (hour <= endTime) {
                        remain = numGrid - curgrid;
            if (remain <= 0) {
                                /* Continue with the grid points for the next hour  */
                                curgrid = 0;
                                hour++;
                                continue;
            }
            allocated = sizeSmallChunks;
            if (problemSizeForProcess[curp] > 0) allocated = sizeSmallChunks - problemSizeForProcess[curp];
            if (remain < allocated) allocated = remain;   /* Allocate remaining to the last process */

            pchunks[curchunk].hour = hour;
            pchunks[curchunk].process = curp;
            pchunks[curchunk].startPoint = curgrid;
            pchunks[curchunk].endPoint = curgrid + allocated -1;
            problemSizeForProcess[curp] += allocated;
            curgrid = pchunks[curchunk].endPoint+1;
            curchunk++;
            if (problemSizeForProcess[curp] >= sizeSmallChunks) curp++;     /* Full size has been allocated to the current process */
                        if (curp >= numProcess && curgrid < numGrid) {
```

```
                                                        /* To handle truncation errors */
                                                        curp--;
                                                        problemSizeForProcess[curp] += numGrid - curgrid;
                                                        pchunks[curchunk-1].endPoint = numGrid-1;
                                                        curgrid = numGrid;
                        }
        }

        /* For debugging */
        if (debug) {
                int pj;
                printf("#\t Process \t Hour \t Start \t End \n");
                for (pj=0; pj < numProblemChunks; pj++) {
                        printf("%d\t %d\t %d\t %d\t %d\n", pj, pchunks[pj].process, pchunks[pj].hour, pchunks[pj].startPoint,
pchunks[pj].endPoint );
                }
        }

        return 1;
}
```

Appendix 2. Main routine of the lighting energy computation

```
/*
        The main routine for lighting power calculations
        Arg1 proc - the process index for the current proces
*/
int startLightingPowerCalculations(int proc) {
        int j,k, hour;

        decomposeProblem();     /* Creates problem chunks pchunks[]  */

        /* Stage 1 - model creation. Done only by proc 0 since the model files need to be manipulated  */
        if (proc == 0) {
                        /* Compile all the models  */
                        compileModels();
        }

        /* Stage 2 - Wait for notification of model creation and then start solving the problem chunk */
        for (j=0; j < numProblemChunks; j++) {
                if (pchunks[j].process != proc) continue;

                if (proc == 0)  {
                        /* Solve the problem chunk without waiting for anyone  */
                        solveProblemChunk(proc, j);
                } else {
                        receiveNotificationModelReady(proc, pchunks[j].hour);
                        /* The above function blocks until the notification is received.  */
                        solveProblemChunk(proc, j);
                }
        }


        /* Stage 3 - Final calculations done only by the first process */
        if (proc == 0) {
                for (j=0; j < numProblemChunks; j++) {
                        if (pchunks[j].process == -1) continue;
                        if (pchunks[j].process == 0) continue;
                        /* Waits until notification is received that the problem chunk is completed */
                        waitForResult(pchunks[j].hour, pchunks[j].process);
                }
                postProcess();
        }

        return 1;
}
```

```
int compileModels() {
        int i, j,k, hour;

        for (hour = startTime; hour <= endTime; hour++) compiledHour[hour] = 0;

        /* Compile the first model for each process */
        for (k=1; k < numProcess; k++) {
                for (i=0; i < numProblemChunks; i++) {
                        if (pchunks[i].process != k) continue;
                        hour = pchunks[i].hour;
                        if (compiledHour[hour] ) continue;
                        compiledHour[hour] = 1;

                         createModelForHour(0, hour);
                         /* Now send notifications to all who use the model for this hour */
                         for (j=0; j < numProblemChunks; j++) {
                                 if (pchunks[j].process == 0 || pchunks[j].process == -1) continue;
                                 if (pchunks[j].hour != hour) continue;
                                 sendNotificationModelReady(pchunks[j].process, pchunks[j].hour);
                         }
                        break;
                }
        }

        /* Now compile all the remaining models  */
        for (hour = startTime; hour <= endTime; hour++) {
                if (compiledHour[hour] == 1) continue;
                compiledHour[hour] = 1;
                createModelForHour(0, hour);
                         /* Now send notifications to all who use the model for this hour */
                for (j=0; j < numProblemChunks; j++) {
                        if (pchunks[j].process == 0 || pchunks[j].process == -1) continue;
                         if (pchunks[j].hour != hour) continue;
                         sendNotificationModelReady(pchunks[j].process, pchunks[j].hour);
                }
        }
        return 1;

}
```

Appendix 3. Synchronization using MPI

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

/* MPI version of the notification function */
int sendNotificationModelReady_MPI(int toproc, int hour) {
        int tag = 50 + hour;
        char message[100];      /* Storage for message  */
        int len;                /* Length of the message        */
        int dest;
        sprintf(message, "model %d", hour);
        dest = toproc;
        len = strlen(message) + 1;      /* including '\0'  */
        MPI_Send(message, len, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        return 1;
}

/* MPI version of the notification function */
int receiveNotificationModelReady_MPI(int proc, int hour) {
        char filename[256];
        char folder[256];
        FILE *in;

        char message[100];      /* Storage for message  */
        int tag = 50 + hour;
        int source = 0;
        MPI_Status status;      /* Return status for receive  */

        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);

        /* Still check if the octree file is readable */
        sprintf(folder, "run-%d-%d", hour, proc );
        sprintf(filename, "%s/octrees/zone.oct", folder);
        in = fopen(filename, "r");
        if (in != NULL) {
                /* The octree file exists already. Return */
                fclose(in);
                return 1;
        }

        /* Wait till the file is available  */
        waitForResultFile(filename);
        return 1;
}
```