Scheduling Independent Tasks with Migration

Deepak Sarda

Submitted to the SMA Office in Partial Fulfillment of the
Requirements for the Degree of

Master Of Science in High Performance Computation For Engineered Systems

SINGAPORE-MIT ALLIANCE

July 2004

Abstract

A multicellular simulation software is being designed at the Bioinformatics Institute. The intracellular biological events of an individual cell are modelled and computed as a process instance. Several such process instances are executed on a parallel cluster of computers to simulate a multicellular population. To ensure best possible simulation runtimes, the software needs to effectively schedule these process instances on the cluster. The present work implements a scheduler module which makes process scheduling decisions. Different scheduling strategies are considered and their relative performance is compared.

Acknowledgement

The work on this project has been an interesting, educational and often challenging experience. I wish to take this opportunity to thank all those people who made this work possible and an enjoyable learning experience for me.

First of all, I wish to express my gratitude to Dr Andrew Goryachev and Dr Wang Jian-Sheng for giving me a chance to work on this project.

I am greatly indebted to Dr. Francis Tang for his continuous guidance and advice throughout the course of this work.

I also wish to thank my friends from the SMA class of 2003-2004 for sharing experiences and knowledge during this delightful year.

Finally, I wish to thank my family for their unceasing support.

I dedicate this thesis to my parents.

Contents

1 Introduction **7**

1.1 Overview 7

1.2 Motivation 8

1.3 Objectives 9

1.4 Organization of Report 9

2 Simulation Framework **10**

2.1 Cells 10

2.2 Master 11

2.3 Scheduler 11

3 Scheduler **14**

3.1 Uniform Tasks 15

3.2 Uniform Runtime 16

3.3 List Scheduler 17

3.4 Implementation 19

4 Testing Framework **21**

5 Results **24**

5.1 Constant population 25

5.1.1 Fully efficient processors 25

5.1.2 Two slow processors 25

5.1.3 One dead processor 26

5.2 Explosive population 27

5.2.1 Fully efficient processors 27

5.2.2 Two slow processors 27
5.2.3 One dead processor 28
5.3 Irregular population 29
5.3.1 Fully efficient processors 29
5.3.2 Two slow processors 30
5.3.3 One dead processor 31

6 Conclusions **32**

References **34**

List of Figures

5.1 The Uniform Times and Uniform Tasks curves overlap 26
5.2 The List Scheduler regains efficiency 26
5.3 The List scheduler again outperforms the others 27
5.4 All schedulers perform equally well 28
5.5 The List and Uniform Runtime schedulers regain efficiency 28
5.6 The List scheduler is markedly better 29
5.7 All schedulers perform equally well 30
5.8 The List scheduler again outperforms the others 30
5.9 The List and Uniform Runtime schedulers regain efficiency 31

Chapter 1

Introduction

1.1 Overview

The Systems Biology Group at Bioinformatics Institute is developing computational models to study complex cooperative behaviour in multicellular populations. The software platform being developed for this study consists of a core simulation program which imitates a single cell's behaviour and then several instances of this program are executed in co-ordination to study multicellular behaviour. Considering the parallel nature of the problem and the computationally intensive nature of the simulations involved, this software platform is being designed to run on a parallel cluster of computers.

A computer cluster is a group of computers, connected by a high speed communications network, that work together as a parallel computer. Such clusters are built chiefly with commodity hardware and hence are able to provide a very good price to performance ratio. Often, the individual computers comprising a cluster are not full fledged computers in the traditional sense. They consist of just a microprocessor, associated memory and a network interface, doing away with the display unit and other peripherals. As such, the term *processor* is often used to describe such a stripped down computer. A *master* computer with permanent

storage and input/output capabilities is used to control and coordinate all the other computers.

When writing programs for such clusters, the traditional sequential programming approach is no longer useful. The parallel programming approach involves:

- Decomposing the problem into parts
- Distributing the parts as tasks which are worked on by multiple processors simultaneously
- Coordinating work and communications of those processors

Bringing things into context, for the multicellular simulation software, the individual cell simulation instances are the parts of the problem. Since the biological events occurring within a cell are to a great extent independent of other cells, these parts can be executed in parallel on different processors. Finally, the results of the individual simulations need to be gathered and processed to determine the changes in the cells and the environment.

1.2 Motivation

The tasks allocated to different processors in a cluster may not be homogeneous. Thus, one task may take longer than the others to complete which means some processors will remain idle until this task completes and the next set of tasks arrive. Also, the number of tasks is usually much greater than the number of available processors. In such cases, several tasks are allocated for execution to each processor and these tasks are then executed sequentially on the individual processors. Therefore, if a processor finishes its pool of tasks before other processors, it will remain idle for some time.

A computing cluster being a shared resource, it is important that task allocation and scheduling be done in a manner which minimizes idle processor time and makes optimal use of the resource. Effective use of the computational resource will automatically ensure faster simulation runtime.

1.3 Objectives

The purpose of the present work is to develop a scheduler module for the multicellular simulation software. Such a module needs to take into consideration several factors like existing processor load, new task creation requests, performance of processors, etc., while making task scheduling decisions. Needless to say, there can be several different approaches to this scheduling problem and these different scheduling strategies will yield different results.

Thus, different scheduling algorithms have to be developed and implemented and their performance has to be compared for various simulation scenarios. Also, the scheduler module needs to be integrated into the main software. Hence, a clean and consistent programming interface is to be developed for the module.

1.4 Organization of Report

In Chapter 2, a description of the various elements involved in the simulation is presented. Section 2.1 describes the cell and its associated computational processes while Section 2.2 outlines the role of the Master process in coordinating the multicellular simulation. Finally, Section 2.3 specifies the role of the scheduler module in detail.

Chapter 3 begins with a brief introduction to the theory behind the task scheduling problem. Subsequent sections desribe the three scheduling algorithms implemented in the present work. The chapter ends with a note on the implementation details of the module.

In Chapter 4, the testing framework used to compare the performance of the different scheduling algorithms is outlined while Chapter 5 details the results of the comparative testing.

Chapter 6 wraps up with inferences drawn from the performance estimation and presents suggestions for effective use of the scheduler module.

Chapter 2

Simulation Framework

2.1 Cells

In the multicellular simulation software, the basic simulation unit corresponds to a single *cell*. The intracellular environment of a cell hosts a wide variety of enzymatic reactions, diffusion events and polymerization. These biological events are transformed into a computational framework using several distinct modelling strategies. These strategies are encompassed as different *computational models* in a core cell simulation program.

Simulation of a cell's biological processes therefore involves executing a process instance of the core cell simulation program. To simulate multiple cells, several such process instances of the simulation program are executed as tasks on the computing cluster. The entire simulation lifetime is broken down into discrete units which we refer to as *time-steps*. During a time-step, there is no intercellular interaction and each cell's simulation runs independent of other simulations.

The environment around the cell affects its life processes. The environment is described by chemical factors such as enzyme concentrations and mechanical factors such as stresses induced by neighbouring cells. At the start of a time-step, each cell is provided information about the current state of the cellular

environment. During the time-step, this information is processed in accordance with the computational model being used to describe the cell's life processes. The simulation over the time-step causes changes withing the cell as well as in its environment. Intracellular changes include a change in the nature of the cell - captured by a change in the computational model describing the cell. Further, a cell could decide to die or divide/replicate itself at the end of a time-step. Physical growth and enzyme excretion are examples of changes that a cell can cause in its environment.

2.2 Master

As described in Section 2.1, the environment affects the cells and vice versa. To collate and propogate the environmental state, a *Master* process is used. At the end of each time-step, the Master process collects results of that time-step's simulation run from each of the cell simulations. It analyzes this data and makes appropriate changes to the environment. It also notes request from cells to die or replicate themselves and takes needed action i.e., cleans up the dead cells and creates new copies of cells requesting replication. It then feeds the current environment state to each of the cells and starts the simulation run over the next time-step.

Cleaning up dead cells involves freeing memory and clearing task queues. Making copies of cells involves copying memory associated with the cell being replicated (parent cell) and creating new instances of the core simulation program.

In order to start process instances to represent newly created cells, the Master process needs to decide which processor(s) of the computating cluster these new instances will be executed on. This decision is provided by the *Scheduler*.

2.3 Scheduler

The *scheduler* is the module which decides which cell simulation process instance runs on which processor of the computating cluster. Here, it must be highlighted

that the simulation run for a new time-step cannot begin until the results from the previous time-step are collated from each of the existing cell simulations by the Master process. This means that a slow processor which hasn't processed all cell simulations assigned to it will hold back the advance of the overall simulation. Thus, the objective of the scheduler is to identify (or predict) the processor which takes the maximum amount of time to complete all the simulations assigned to it and try and *minimize this maximum time*.

To aid it in making allocation decisions, the scheduler module is provided with performance data from the previous time-step. This performance data includes the time each individual cell simulation took to complete and the time that each processor took to finish all the simulations assigned to it. This time is the wall clock time (real time) and is not related to the time-step of the simulation (virtual time).

The run time of a processor will generally be greater than the cumulative run time of all individual cells assigned to it. This is because of the shared resource nature of computational clusters as was mentioned in Chapter 1. Existence of other computational tasks on a processor which are unrelated to the cell simulation software could result in queuing of the cell simulation tasks resulting in a longer run time. As will be desribed in Chapter 3, the cell simulation run times and the processor run times taken together provide a measure of the processing load on the processor.

The scheduler is also provided information regarding the requests to die or replicate by the cells. This data lets the scheduler know resources on which processor are being freed and how many new tasks need to be created.

Based on these inputs, the scheduler returns the following decisions to the Master.

- The processor on which each of the newly created cells will be executed.
- *Additionally*, if some cell simulations need to be migrated to different processors with a view to increasing overall simulation performance.

Of note here is the fact that each task has information associated with it and hence, migrating a task from one processor to another requires time. This time chiefly depends on the number of bytes of information being transferred between

processors and is usually significant because of the relatively slow speeds of the network interconnects used in computational clusters. Therefore, this time of transfer needs to be taken into account by the scheduler when making schduling decisions which involve migrations.

Chapter 3

Scheduler

In terms of scheduling theory, the task assignment problem described in Section 2.3 is that of scheduling a set of n independent tasks on m unrelated processors. The tasks are independent in the sense that

1. There are no precedence constraints such as 'Task A must complete before Task B begins'

2. A task does not depend on data from another task during execution

Point one implies that the time of start of execution of a task depends just on resource availability while point two implies that the task's completion time depends just on the nature of the task and the processor's efficiency. For the multicellular simulation program, the cell simulation tasks are indeed independent. As described in Section 2.1, during a time-step, the cells exchange no data and are completely independent in nature.

The processers used to complete the set of tasks could be *identical*, *uniform* or *unrelated*. If the processors are identical, they take the same amount of time to complete a given task. If they are uniform, then the time of completion of a task is just a function of the speed of each processor. In the unrelated processors case, although every processor can handle every task, the time required by a processor to complete a given task is a function of both the processor and the task.

Clearly, the present problem involves unrelated processors since the time required for finishing a cell simulation depends firstly on the cell state and computational model being used for the cell and secondly on the efficiency of the processor to which it has been assigned.

As described in Section 2.3, the objective that we are trying to minimize is the maximum task completion time on any processor, also called the *makespan*. For a variable number of unrelated processors m, the decision problem of determining whether a set of independent tasks can be assigned with finishing time smaller than a given bound has been shown to be NP-complete [4]. This means that it is unlikely that an algorithm with polynomial runtime exists for the problem being considered.

For the case of fixed number of processors, Horowitz and Sahni [3] have proven that for any $\epsilon > 0$, an ϵ-approximate solution can be computed in $O\left(nm\left(nm/\epsilon\right)^{m-1}\right)$ time, which is polynomial in n and $1/\epsilon$ if m is constant. But as is clear, even for a small computing cluster of 16 processors, this polynomial time algorithm is of little practical value.

Thus, for the scheduler module, we use heuristic approaches to solve the scheduling problem. In this chapter, we present a description of the different heuristics algorithms that have been implemented.

As a matter of notation, the execution of tasks by processors is modelled by the notion of an assignment function. An *assignment function* A is a map $A : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ from the set of tasks to the set of processors. If $A(t) = p$, we say that task t is assigned to processor p. Clearly, under the map, each task is assigned to one and only one processor and no task is left unassigned.

3.1 Uniform Tasks

This is an extremely simple scheduling strategy whereby we determine the number of tasks a processor is currently assigned and then assign new tasks to the processor with the least assigned tasks. In effect, we are trying to distribute the tasks equally among processors. As the algorithm below illustrates, there is no provision for migration of tasks using this method.

Algorithm

Step 1 Set $sum_p = P_p$ for $p = 1, \ldots, m$ (P_p is the number of tasks already assigned to pth processor from the previous time-step)

Step 2 **while** not all tasks are assigned **do**

begin

Step 3 Find a value of p such that sum_p is minimum

Step 4 Pick an unassigned task t and assign it to p, i.e. set $A(t) = p$

Step 5 Set $sum_p = sum_p + 1$

end

3.2 Uniform Runtime

This is a more sophisticated approach than the Uniform Tasks algorithm described in Section 3.1. We first try and estimate the expected runtime of each unassigned cell. For this, we assume that the runtime of a task t, denoted $\lambda(t)$, is a function of the computational model (Section 2.1)that describes the cell. We group cells with the same model describing them and compute their average runtime λ_j (j is the model) using $\lambda(t)$ values from the previous time-step. These $\lambda(t)$ values for the tasks in the previous time-step can be determined during execution using standard timing functions. For the first time-step when no data from previous time-step is available, we just assume that $\lambda_j = 1 \forall j$.

We define the efficiency of a processor $\eta(p)$ as the ratio of the cumulative runtimes of all cells assigned to the processor to the total runtime of the processor ($\rho(p)$), i.e.

$$\eta(p) = \frac{\sum\limits_{A(t)=p} \lambda\left(t\right)}{\rho\left(p\right)}$$

As detailed in Section 2.3, this ratio is frequently less than unity since presence of other tasks on the processor that are unrelated to the cell simulation software can cause delay in execution of cell simulation related tasks. Thus, although the processor finishes execution of cell simulation tasks in $\sum\limits_{A(t)=p} \lambda\left(t\right)$, the effective

runtime for the time-step is $\rho\,(p)$.

Further, this algorithm too doesn't allow for migration of tasks. The task assignment is done according to the following algorithm.

Algorithm

Step 1 Compute λ_j, the average expected task runtimes

Step 2 Compute the processor efficiencies $\eta(p) = \frac{\sum\limits_{A(t)=p} \lambda(t)}{\rho(p)}$ for $p = 1, \ldots, m$

Step 3 Set $sum_p = \rho\,(p)$ for $p = 1, \ldots, m$

Step 4 **while** not all tasks are assigned **do**

begin

Step 5 Find a value of p such that sum_p is minimum

Step 6 Pick an unassigned task t and assign it to p, i.e. set $A(t) = p$

Step 7 Set $sum_p = sum_p + \frac{\lambda_j(t)}{\eta(p)}$

end

3.3 List Scheduler

This algorithm is an adaptation of the often used list scheduling algorithm [1]. A simple list scheduling algorithm begins by building a list of tasks to be executed, either in an arbitrary manner or according to some heuristic. The algorithm proceeds by finding the processor which will first complete the tasks it has already been assigned and assigning to that processor the next task on the list.

The algorithm we consider here has been proposed by Davis and Jaffe [2] and differs from the simple list scheduling algorithm in two ways. First, a separate list is constructed for each processor in order to reflect their different efficiencies on different tasks. Second, if a processor is very inefficient for the next task on its list, the task is not assigned to the processor; rather, the processor is taken out of the pool of available processors.

This algorithm is at most $2.5\sqrt{m}$ times worse than optimal in the worst case and has a running time of $O(mn \log n)$ where m is the number of processors and n is the number of tasks.

For this algorithm, we first need to define a task system matrix μ where the element $\mu(t,p)$ is the time required by a task t to complete execution on a processor p. This task system matrix reflects the unrelated processors nature of the present problem, as described earlier in this chapter. We continue with the definitions of λ_j, the average task runtime and $\eta(p)$, the processor efficiency from Section 3.2. We choose to calculate μ as

$$\mu(t,p) = \frac{\lambda_j(t)}{\eta(p)} + (1 - \delta_{p,q}) \times C$$

where q is the processor to which t is currently assigned, C is the cost of migration (in time units) from processor q to p and δ is the *Kronecker Delta* function defined as

$$\delta_{p,q} = \left\{ \begin{array}{ll} 1 & \text{if} p = q \\ 0 & \text{otherwise} \end{array} \right\}$$

This choice of μ allows the use of the Davis and Jaffe algorithm to allow for migration of tasks from inefficient processors. If the benefits from migrating are greater than the cost of migration, the algorithm will try to migrate the task. For new tasks (from cell replication), we initially assign them to their *parent* cell's processor and let the algorithm decide if they are best created on the same processor as their parent or on a different processor.

It should be mentioned that when we call a processor *inefficient* (low $\eta(p)$), it is only with respect to our cell simulation software. A processor may be very fast and efficient at processing tasks allocated to it but if it is loaded with tasks unrelated to our cell simulation software, that efficiency is practically useless. We perceive a low processor efficiency and try and migrate some of our tasks to a processor willing to devote more computational cycles to our simulation software.

The *best time* of the tth task, denoted $b(t)$, is the smallest of the m values $\mu(t,p)$. The efficiency of the pth processor on the tth task, denoted $\mathrm{ef}(t,p)$, is $b(t)/\mu(t,p)$.

Algorithm

Step 1 For $t = 1, \ldots, n$, find $b(t) = min_{1 \leq p \leq m} \mu(t,p)$

For $t = 1, \ldots, n$, $p = 1, \ldots, m$, find $\mathrm{ef}(t,p)$=$b(t)/\mu(t,p)$

For $p = 1, \ldots, m$, create a list of tasks $t = 1, \ldots, n$

sorted in non-increasing order of ef(t, p)

Set $sum_p = 0$ for $p = 1, \ldots, m$ (sum_p is the finishing time of the pth processor under the tasks thus far assigned to it)

Designate all processors as "active" and all tasks as "unassigned"

Step 2 **while** not all tasks are assigned **do**

begin

Step 3 Find a value of p such that sum_p is minimum among active processors

Step 4 Find the next unassigned task t on p's list of tasks

Step 5 **if** t does not exist or **if** ef$(t, p) < 1/\sqrt{m}$ **then** mark p as inactive

else Define $A(t) = p$. Designate t as being assigned.

Set $sum_p = sum_p + \mu(t, p)$

end

3.4 Implementation

One of the main considerations in the design of the scheduler is that the implementation should be modular allowing for easy swapping of the algorithms described above. Also, code should be reused between algorithm implementations as far as possible. The object-oriented programming style lends itself naturally to addressing these issues.

The scheduler module has been implemented in C++ since the language is object-oriented leading to cleaner code and faster development time. Further, through the Standard Template Library, it provides excellent support for advanced data structures like *vectors* and *maps* which have been extensively used in this implementation.

If we look at the problem, it is clear that some aspects like communication of data between the main module and the scheduler module are identical irrespective of the algorithm under consideration. Thus, these details can be implemented in a *base class* while the algorithm specific details can be implemented in different classes derived from this base class.

The base class contains members which capture the state of all cells and

processors. It provides methods which communicate runtime statistics and cell creation requests from the main module to the scheduler module and then the task scheduling decisions from the scheduling module back to the main module.

The derived classes corresponding to the different algorithms contain just the algorithm specific implementation details. Since the derived classes inherit the base class members and methods, the algorithms can easily act on the state information captured in the base class members.

Splitting the scheduler according to this class hierarchy allows for the provision of a consistent programming interface to the main module. The same function calls and data structures can be used in the main module irrespective of the algorithm chosen to schedule tasks.

Chapter 4

Testing Framework

To test the working and evaluate the performance of the different scheduling algorithms described in Chapter 3, test case data is needed. Since the multicellular simulation software platform isn't yet in a stage where it can provide test data, a program which creates pseudo random data points under various scenarios was written to generate these test cases.

To recapitulate from Section 2.1, the simulation model being considered starts at an initial time with a certain cell population. The simulation proceeds in small slices of (virtual) time which we call *time-steps*. Within each time step, certain biological events occur inside each cell and one of the results of these events is a decision regarding the fate of the cell at the end of the current time-step.

There are three possible fates for a cell:

1. It may choose to replicate itself into a clone cell.

2. It may choose to die.

3. It may choose to live into the next time step.

For the purpose of generating test data, we associate a finite probability with each of these scenarios. By suitably modifying the probabilities, it is possible to simulate a cell population which is stable, growing exponentially, etc. For

example, if we choose zero probability of replication or of death, the population remains constant at the initial level. If the probability of replication is chosen to be considerably higher than the probability of death, a population experiencing exponential growth can be simulated.

As described in Section 2.1, the exact nature of the actual biological processes carried out within a cell depend on the computational model that describes the cell. For our purposes, it is unnecessary to look at the details of the biological events occurring. We reiterate the assumption made in Section 3.2 that the runtime of a cell simulation program is a function of the computational model that describes the cell.

For the purpose of generating test case data, we assume that there are three such computational models to describe the cells. We also make the assumption that the simulation runtimes for *similar* cells (cells being described by the same computational model) follow a gaussian distribution pattern. Thus, we associate with each of the three computational models, a unique gaussian distribution of runtimes.

We further assume that all the cells at initial time are described by the first of the three models. At the end of a time step, an additional decision is made by each cell regarding a change of model. According to a finite probability, a cell may switch to a different model. This allows for a measure of control over the evolution of the nature of the cell population.

The test case generation program takes as input the different probabilities described above, the initial population size, the maximum population size and the maximum number of time steps. We make use of the functions provided by the *GNU Scientific Library* to generate the various probability distributions. All of the functions utilize one global seed value; all other inputs being same, a different global seed will generate a different test case.

The output is a test case data file which describes the evolution of the cell population. It tracks the life of each cell, the decisions it makes and the simulation runtime it needs for each time-step.

As discussed in Sections 2.3 and 3.3, the load on a processor may not comprise entirely of cell simulation software related tasks. Hence, the efficiency of the processor as perceived by us will be less than one. To model this work load of the

processors, we use a data file which tracks the efficiencies of each processor ($\rho(p)$) at each time-step, the best efficiency being one. A wrapper program uses these test case data files and runs the scheduling module for the different algorithms. It collates the results and creates output for the required performance metrics.

Chapter 5

Results

To compare the performance of the different algorithms detailed in Chapter 3, we need a performance metric. We choose this metric to be the overall simulation efficiency at each time-step, denoted E_i, where i is the time-step index. We define this efficiency as

$$E_i = \frac{\sum \lambda\left(t\right)}{M \times max\left\{\rho\left(p\right)\right\}}$$

where $t \in \{1, \ldots, n\}$, $p \in \{1, \ldots, m\}$ and M is the total number of processors.

As was described in Chapter 4, a test case comprises of two sets of data. First, the cell population evolution data and second, the processor efficiency data. For the former, we consider the following scenarios:

1. Constant population
2. Explosive population growth
3. Irregular population

For the processor efficiencies, we fix the number of processors to be four and consider the following scenarios:

1. All processors working at full efficiency.

2. Two processors slow down to 60% efficiency, one when the simulation is 25% complete and another mid-way through the simulation; the former processor regains full efficiency at 75% of simulation completion.

3. One processor slows down to 5% efficiency (practically dead) mid-way through the simulation.

Here, simulation completion refers to the number of time-steps for which test case data was generated.

5.1 Constant population

5.1.1 Fully efficient processors

When the simulation starts, the initial task allocation to the four processors is done identically by all three schedulers. Since the *Uniform Tasks* and *Uniform Runtime* schedulers do not support migration of tasks, without the creation of new cells, these schedulers do not have any more decisions to make. Hence they perform identically.

For the case of all processors performing equally well [Fig 5.1], the *List* scheduler differs from the other two since it tries to migrate tasks based on the variations in cell runtimes.

5.1.2 Two slow processors

Since the *List* scheduler allows for migration, when the efficiency drops in two processors [Fig 5.2], it immediately moves tasks to the other processors and we see the immediate improvement in the overall efficiency. The other schedulers, unable to do any migrations, show a markedly decreased overall efficiency.

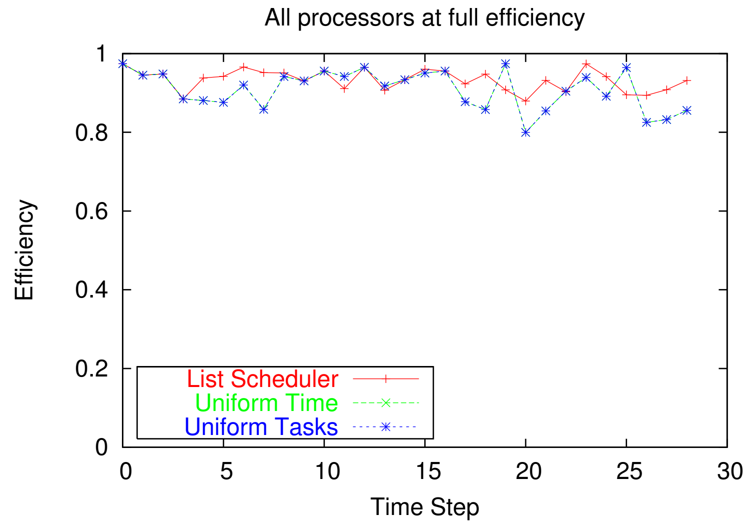

Figure 5.1: The Uniform Times and Uniform Tasks curves overlap

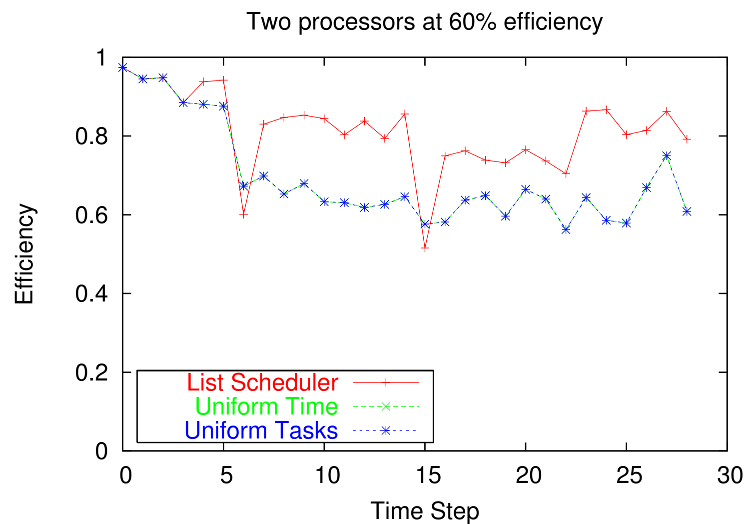

Figure 5.2: The List Scheduler regains efficiency

5.1.3 One dead processor

For the case when one processor is effectively dead [Fig 5.3], the situation is analogous to that of the earlier case but the performance difference between the schedulers is more drastic.

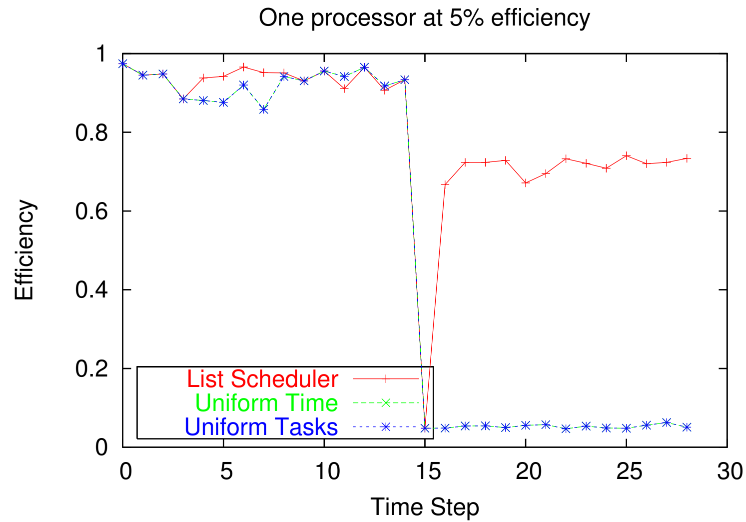

Figure 5.3: The List scheduler again outperforms the others

5.2 Explosive population

5.2.1 Fully efficient processors

When all the processors are working at full efficiency, we find that all the schedulers work equally well [Fig 5.4].

5.2.2 Two slow processors

When two processors slow down, the *Uniform Tasks* scheduler's performance drops but the other two schedulers regain efficiency quickly [Fig 5.5]. Although, the *Uniform Runtime* scheduler can't migrate tasks, the death of old cells and the rapid influx of newly created cells allows it to effectively schedule new tasks and counter the marginal decrease in processor effciencies.

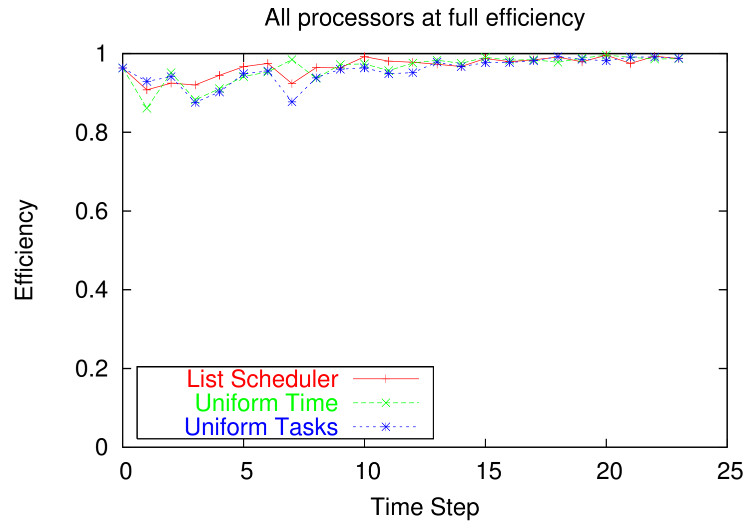

Figure 5.4: All schedulers perform equally well

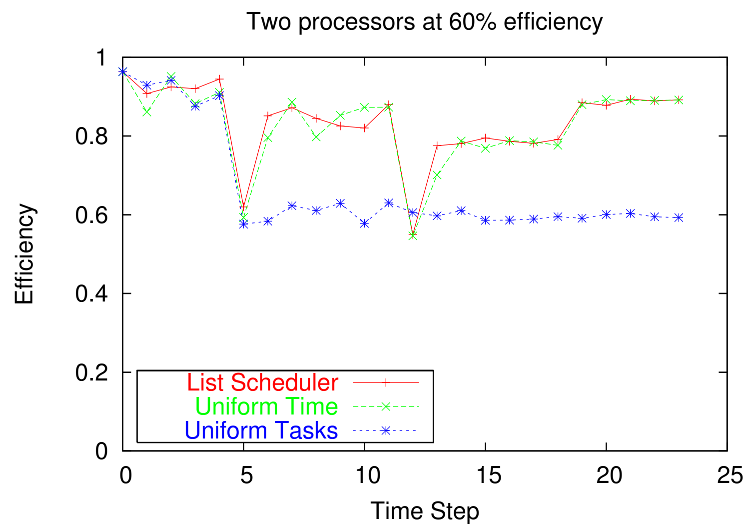

Figure 5.5: The List and Uniform Runtime schedulers regain efficiency

5.2.3 One dead processor

In this case, we find that when one processor becomes extremely slow, the *List* scheduler immediately migrates tasks and regains efficiency [Fig 5.6]. But unlike the previous case of two slow processors, the *Uniform Runtime* scheduler does

not perform well. The drop in efficiency of the processor is too drastic and the scheduler takes several time-steps (several new cell creation requests) before reaching an efficiency comparable to the *List* scheduler.

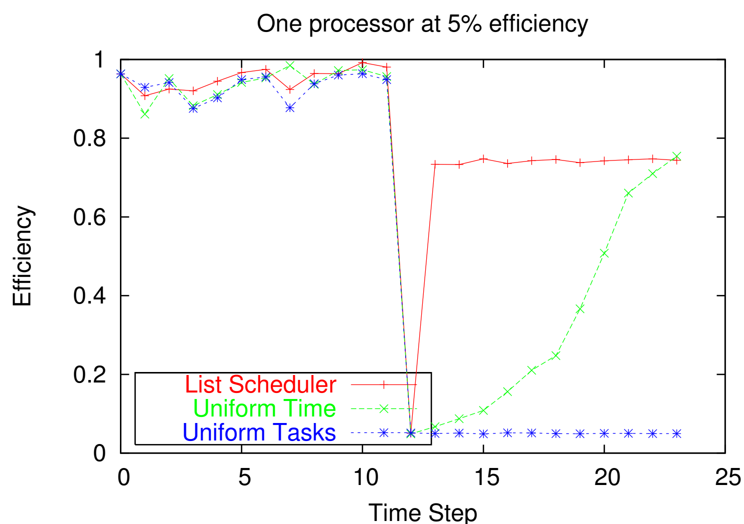

Figure 5.6: The List scheduler is markedly better

5.3 Irregular population

5.3.1 Fully efficient processors

In this test case, although the net population does not increase rapidly, there is a very high rate of change - a great number of cells are dying and a similar number new cells are being formed by cloning in each time-step. When all the processors are working at full efficiency, we find that the overall efficiency achieved by each of the schedulers reflect the irregular nature of the cell population [Fig 5.7].

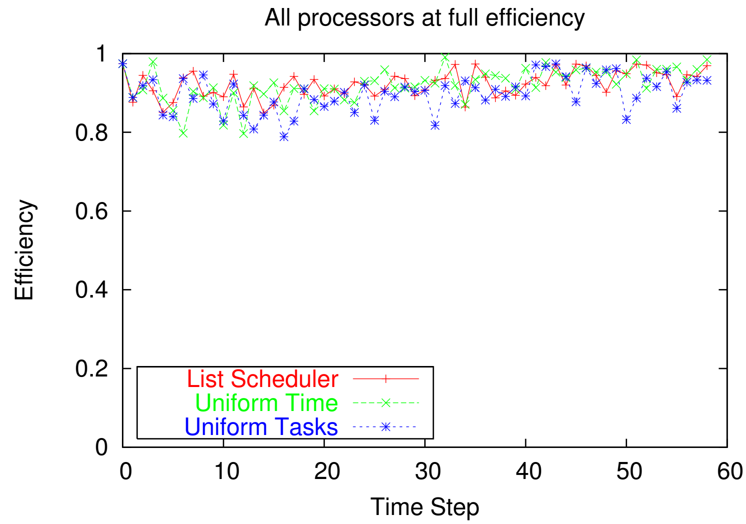

Figure 5.7: All schedulers perform equally well

5.3.2 Two slow processors

In this case [Fig 5.8], the results are similar to those obtained for the case in section 5.2.2, although not as uniform. A similar explanation holds too.

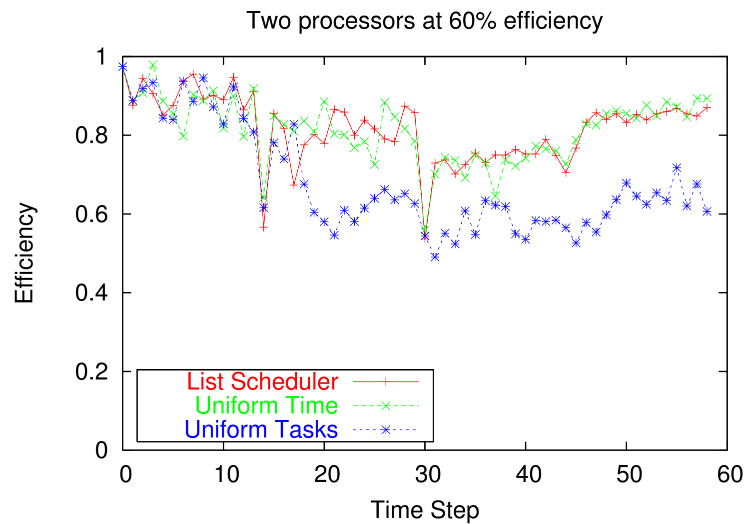

Figure 5.8: The List scheduler again outperforms the others

5.3.3 One dead processor

As was the situation in the case studied in section 5.2.3, the performance of the *Uniform Runtime* scheduler drops greatly but this time, the regaining of efficiency is more erratic [Fig 5.9].

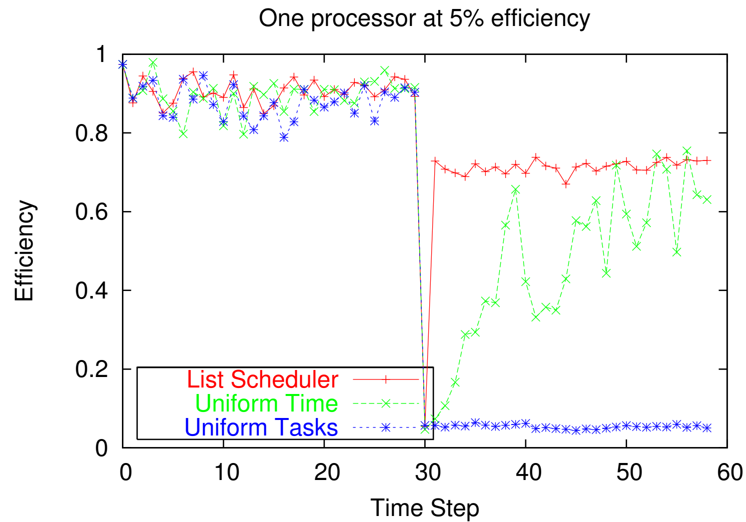

Figure 5.9: The List and Uniform Runtime schedulers regain efficiency

Chapter 6

Conclusions

From the test case results seen in Chapter 5, we find that the *Uniform Tasks* scheduler shows good performance only for the ideal case when all the processors are working at full efficiency. Even in such a case, since the *Uniform Runtime* scheduler is only marginally slower than the *Uniform Tasks* scheduler, the use of the former is recommended as it gives better results for slight deviations from the ideal situation.

We also find that the *List* scheduler performs well for all the cases. The *Uniform Runtime* scheduler, although not uniformly as good as the *List* scheduler, performs remarkably well in certain situations such as when the population is growing rapidly with an influx of a large number of new cells.

The *List* scheduler is a complex algorithm and its uniformly good performance comes at the expense of significantly higher processing time (especially for large number of tasks) when compared to the *Uniform Runtime* scheduler. In situations like having marginally inefficient processors or an almost constant population, it might be worthwhile to use the quick *Uniform Runtime* scheduler, especially if the actual simulation runtimes are small. Alternatively, some other heuristics may be used, such as using the *List* scheduling algorithm only once every x time-steps and calling the *Uniform Runtime* scheduler at all other time-steps.

To provide this kind of flexibility in choosing the scheduling algorithm at run-

time based on requirements, the original implementation of the scheduler module was modified. While in the original implementation, the scheduling algorithm would have to be specified at the beginning of the simulation, the modified version allows for the specification of the algorithm at each time-step of the simulation.

Bibliography

[1] J. Bruno, E. G. Coffman, Jr., and R. Sethi, *Scheduling independent tasks to reduce mean finishing time*, Commun. ACM **17** (1974), no. 7, 382–387. 3.3

[2] Ernest Davis and Jeffrey M. Jaffe, *Algorithms for scheduling tasks on unrelated processors*, J. ACM **28** (1981), no. 4, 721–736. 3.3

[3] Ellis Horowitz and Sartaj Sahni, *Exact and approximate algorithms for scheduling nonidentical processors*, J. ACM **23** (1976), no. 2, 317–327. 3

[4] J. K. Lenstra, D. B. Shmoys, and E. Tardos, *Approximation algorithms for scheduling unrelated parallel machines*, Math. Program. **46** (1990), no. 3, 259–271. 3