ACCELERATING POPULATION BALANCE MODEL - BASED PARTICULATE PROCESS SIMULATIONS VIA PARALLEL COMPUTING

BY ANUJ VARGHESE PRAKASH

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Chemical and Biochemical Engineering

Written under the direction of

Dr. Rohit Ramachandran

and approved by

New Brunswick, New Jersey

January, 2013

ABSTRACT OF THE THESIS

Accelerating Population Balance Model - based particulate process simulations via parallel computing

by Anuj Varghese Prakash

Thesis Director: Dr. Rohit Ramachandran

The use of Population Balance Models (PBM) for simulating dynamics of particulate systems are inevitably limited at some point by the demands they place on computational resources. PBMs are widely used to describe the time evolutions and distributions of many industrial particulate processes, and its efficient and quick simulation would certainly be beneficial for process design, control and optimization. This thesis is an elucidation of how MATLAB's Parallel Computing Toolbox (PCT), a third-party toolbox called JACKET, and the MATLAB Distributed Computing Server (MDCS) may be combined with algorithmic modification of the PBM to speed up these computations on a CPU (Central Processing Unit), GPU (Graphics Processing Unit) and a computer cluster respectively. Parallel algorithms were developed for three dimensional and four dimensional population balance models incorporating hardware class-specific parallel constructs such as `SPMD` and `gfor`. Results indicate significant reduction in computational time without compromising numerical accuracy for all cases except for the GPU. The GPU seemed promising for larger problems despite its limitations of lower clock speeds and on-board memory compared to the CPU. Evaluations of the speedup and scalability further affirm the algorithms' performance.

Acknowledgements

To try and thank everyone who made my life and work here at Rutgers - this thesis being its culmination - a memorable one, is decidedly daunting. Nevertheless, I will try to try. Foremost, my advisor, Dr. Rohit Ramachandran, for his encouragement and invaluable suggestions which have guided me since day one of this endeavour, and also other members of my thesis committee: Dr. Preetanshu Pandey, Prof. Marianthi Ierapetritou and Prof. Meenakshi Dutt. My sincere gratitude to PhD students Anwesha Chaudhury, Dana Barrosso and Maitraye Sen for the insightful discussions, instrumental in giving shape to my thoughts and ideas. Special thanks to Dr. Atul Dubey for introducing me to the esoteric art of PC building and graciously allowing me to sit in his impressive office. To Tom McHugh, Mathworks, for providing and extending the MDCS license without which this work would remain incomplete. Lastly, the people who made my stay at Rutgers a wonderful one - my fellow group members, room-mates and dear friends in and out of the CBE family - thank you for the memories, I am truly indebted.

Dedication

"Whom have I in heaven but thee? and there is none upon earth that I desire beside thee. My flesh and my heart faileth: but God is the strength of my heart, and my portion for ever."

- Psalms 73:25,26

This work is dedicated to my Lord and Savior, Jesus Christ, for giving meaning to my life; and, of course, to my wonderful, one-of-a-kind family.

Table of Contents

List of Figures

Chapter 1

Introduction

In the domain of particulate processes, computer aided modeling and simulation now form an indispensable part of research and development activities within industry and academia. It accelerates process development and design, enables equipment sizing and rating, and finally, facilitates process control and optimization.[1] This is due in no small part to significant leaps made in computer hardware architecture over the last few decades leading to the rise of faster and more efficient CPUs with each new generation. With CPU clock speeds gradually reaching their theoretical limits,[2] processor manufacturers are now resorting to the addition of more cores and incorporation of multi-threading in their CPUs. From a programmer's point of view, this entails rewriting sequential code to distribute data and/or tasks to be evaluated separately on different processing units, a practice known as parallel or concurrent programming.[3] Commonly termed 'parallelization', it involves the breaking down of a single given problem into simpler sub-problems that may be solved by utilizing multiple processing units working concurrently. If factors such as communication between processors, load balancing and data dependency have been accounted for, parallelization yields considerable reduction in application run time. Owing to the discrete nature of granular materials, parallel programming lends itself well to the modeling of particulate processes such as crystallization, granulation, milling and polymerization. These unit operations are fundamental to the manufacture of bulk commercial products like pharmaceuticals, detergents, fertilizers, and polymers. Progress in research on particulate dynamics has been rising steadily over the last few decades[45],[6] notwithstanding the fact that these systems are inherently dynamic in behavior, making them unpredictable, driven by complex micro-scale phenomena.[7] Granulation is one such particle design process where significant advances have been made in modeling and simulation.[8] Approaches for modeling such systems are as numerous as they are varied: Discrete Element Modeling (DEM),[9] population balance modeling (PBM),[10,11,6,12,13,14,15] hybrid models by combining PBM with DEM,[16] PBM with Volume of Fluid (VoF) methods,[17] PBM with computational fluid dynamics (CFD),[18] to name a few. PBMs are more suited to simulate large numbers of particles over extensive time periods due to its semi-mechanistic approach (compared to the relatively more mechanistic approaches of DEM and VoF).[19] It provides a convenient mathematical framework whereby the level of model detail is user specified, depending on the kernel formulations incorporated.[10] Thus, PBM offers a highly efficient way of developing a comprehensive model for simulation, control and optimization of the granulation process.[20]

However, PBM is limited by the computational expenses it incurs in terms of run time and hardware resources. Depending on the complexity of physical phenomena modeled into the system, simulations can take anywhere from a few hours to days to reach completion. This is exacerbated by the fact that computational load increases almost exponentially on increasing dimensionality of the system, leading to longer run times, even in a high-level language environment like MATLAB. MATLAB is one of the preferred languages of development for scientific computing due to the ease with which algorithms can be developed and prototyped, which in turn, is enabled by its array-based semantics, powerful visualization capabilities and subject-specific toolboxes, all encased in an integrated framework [21] . While MATLAB excels on the 'ease of programmability' and 'portability' fronts, it has been found to be lacking in the 'performance' department [22] . This is partly due to the abnormally high memory requirements of modern scientific applications, and partly due to the fact that MATLAB itself consumes a sizable portion of the system memory. In addition, a MATLAB code for a process such as granulation typically has several nested for-loops and multiple operations over large data sets that execute 'serially' or 'sequentially' by default, drastically bringing down the rate of simulation. Further discussion of observed execution bottlenecks in a PBM code can be found in the next section. Although researchers are continuously upgrading their hardware to include the latest CPUs and higher amounts of RAM (Random access memory) in an attempt to improve calculation efficiencies, most of them do not develop codes that fully leverage the parallel processing capabilities of the current generation of multi-core / multi-processor CPUs. Furthermore, due to limitations on the power density that can be supplied, peak CPU clock frequency attainable is restricted (6 GHz for an Intel Core i5 processor[23]).

By parallelizing the simulation, a programmer is able to circumvent the restriction of having to run code sequentially on one core, and thus harness the power of multiple processors within the same machine (parallel) or even a cluster of machines (distributed), saving computational time. MATLAB is a widely used, high-level language within the scientific computing community, and therefore this work will describe code development based on toolboxes (function libraries) supported by it. For parallel programming, two toolboxes are pertinent: the Parallel Computing Toolbox (PCT) and MATLAB Distributed Computing Server (MDCS). Specifics on each toolbox can be found in the next section. In addition to multi-core CPUs, parallelism may be achieved on many-core devices such as the GPU (Graphics Processing Unit) and the Intel Xeon Phi coprocessor. For the purpose of GPU computation, the toolbox from Accelereyes inc. named JACKET was chosen as it clearly outperforms MATLAB's built-in capabilities.[24] There has been some recent work on the parallelization of PBM simulations prior to this study. Gunawan et al (2008)[25] formulated an efficient way of parallelizing High Resolution Finite Volume (HRFV)-solved PBEs (Population Balance Equations) by assigning the operations on particles in the first half of the size range that were more computationally intensive to processors of greater rank, and operations on the other size range half of decreasing load intensity to higher ranked processors. Their strategy enabled efficient load distribution and resulted in near linear speedup. More recently, Ganesan and Tobiska (2011)[26] built upon this work by developing a finite element approach of splitting the PBE dimensionally into spatial and internal coordinates, permitting the problem to be parallelized easily without the need for load balancing.

1.1 Overview

This work aims at developing proper coding techniques to enable parallel and distributed execution on CPUs, GPUs and extend them to clusters with the aid of the PCT, JACKET and MDCS toolboxes respectively. A combination of both data and task parallel styles of programming was followed with regard to individual CPUs and GPUs, while a distributed shared memory approach was chosen for execution on a cluster without the need for load balancing or explicit inter-processor communication. To illustrate the applicability of these techniques, several multi-dimensional PB models of granulation were developed and parallelized to run across each class of hardware. This thesis consists of 6 chapters:

1. 'Introduction' provides an overview of parallel computing with its proposed speedup benefits for population balance model-based simulations. In addition, information on previous work done in the field is also outlined.

2. 'Population Balance Modeling' introduces and explains the fundamentals of population balance modeling, specifically in the context of powder granulation.

3. 'Parallel, distributed and GPU computing' details the various ways of incorporating parallelism based on the problem/hardware combination at hand.

4. 'Model development and parallel programming strategy' lays down the general framework for parallelization, followed a detailed procedure of model development pertinent to each hardware class.

5. 'Case studies: Results and discussion' focuses on execution the parallelized granulation models from the previous section on a multi-core CPU, a GPU and a distributed system of two multi-core machines. This is followed by a statistical analysis of achieved speedup and evaluation of other performance metrics.

6. A summary of this entire work, and a discussion on possible directions to further research in this area are proposed in the final chapter titled 'Conclusions and recommendations for future work'.

Chapter 2

Population Balance Modeling

A general form of the population balance equation (2.1) highlighting the temporal variation of the distribution of one or more intrinsic properties is given below :[27]

$$\frac{\partial F}{\partial t}(\mathbf{x},t) + \frac{\partial}{\partial \mathbf{x}}\left[F(\mathbf{x},t)\frac{d\mathbf{x}}{dt}\right] = \Re_{formation}(\mathbf{x},t) - \Re_{depletion}(\mathbf{x},t) \tag{2.1}$$

where, F is the particle number distribution and x is the vector of internal coordinates used to define the process. $\Re_{formation}$ and $\Re_{depletion}$ represent the net formation and depletion rates of particles occurring from all discrete granulation mechanisms such as aggregation, nucleation and breakage. By convention, PBMs have been described by a single intrinsic property such as particle size.[28]

2.1 Developing a 3 dimensional PB framework

Dependence on solely particle size was found to be inadequate in characterizing variability inherent to the granulation process, and soon thereafter other factors began to find their way into descriptions of PBMs.[29, 30] In addition to granule size, binder content and granule porosity are typically selected as decisive factors in optimizing and controlling the process, as evidenced in current research on granulation.[13] Verkoeijen et al [31] had previously described an efficient way of implementing such a multi-dimensional framework by expressing the intrinsic properties of granules – i.e. the volume of solids s, volume of liquid l, and volume of gas g – as a vector in volume space with three coordinates i.e. s, l and g. Particle internal coordinates (Equation (2.2)) are now represented as:

$$\mathbf{x} = \begin{bmatrix} s & l & g \end{bmatrix} \tag{2.2}$$

where each of these three co-ordinates (s, l, g) comprises unique distributions of phase volume fractions (solid, liquid or gas) of particles belonging to a pre-defined volume class, and can therefore be represented as three separate discretized domains or 'grids', containing the distributions. These grids are composed of 'bins' that represent the volume classes for each phase that constitutes a particle in the population. For instance, the first bin in the solid volume grid represents the particles that have the least solid volume, the next bin contains a fraction of particles with higher solid content and so on. This way, the individual solid volumes of the particles can be represented by allocating them in the corresponding bins. The same principle applies to the other two phase fraction grids, liquid(l) and gas(g). In this document, the term 'Grid size' will be used to refer to the total number of bins in a grid. This discretized approach has two important benefits: (a) it enables decoupling of individual mesoscopic processes like aggregation, consolidation and layering; (b) it improves the numerical solution of the aggregation model due to the mutually exclusive nature of the internal coordinates.[10] Such a 3-D model can now describe changes in the volume distribution of particle volume with respect to time,[6] as shown below:

$$\begin{aligned}&\frac{\partial}{\partial t}F(s,l,g,t)+\frac{\partial}{\partial g}\left[F(s,l,g,t)\frac{dg}{dt}\right]+\frac{\partial}{\partial s}\left[F(s,l,g,t)\frac{ds}{dt}\right]\\&+\frac{\partial}{\partial l}\left[F(s,l,g,t)\frac{dl}{dt}\right]=\Re_{aggregation}+\Re_{breakage}+\Re_{nucleation}\end{aligned}\tag{2.3}$$

where $F(s, l, g, t)$ represents the population density function with $F(s, l, g, t)dsdldg$ referring to the moles of granules with solid volume between s and $s+ds$, liquid volume between l and $l+dl$ and gas volume between g and $g+dg$. The partial derivative term with respect to s accounts for the layering of fines onto the granule surfaces; the term with respect to l accounts for the drying of the binder and the re-wetting of granules; the term with respect to g accounts for consolidation which, due to compaction of the granules, results in a continuous decrease in pore volume and an increase in pore saturation. The terms on the right hand side of the above equation refers to the source terms, namely, aggregation ($\Re_{aggregation}$), breakage ($\Re_{breakage}$) and nucleation ($\Re_{nucleation}$). In the current work, nucleation is not considered to be a significant factor and will be dropped while building the population balance model.

2.1.1 Aggregation

The $\Re_{aggregation}$ terms (Equations 2.4,2.5,2.6) takes into account the formation/depletion of granules due to aggregation, for which the terms have been defined in literature[32] as:

$$\Re_{agg}(s,l,g,t) = \Re_{agg}^{formation} - \Re_{agg}^{depletion}, \tag{2.4}$$

$$\begin{aligned} \Re_{agg}^{formation} = \frac{1}{2}\int_{s_{nuc}}^{s-s_{nuc}} \int_{0}^{l_{max}} \int_{0}^{g_{max}} \beta(s', s-s', l', l-l', g', g-g') \\ \times F(s',l',g',t)F(s', s-s', l', l-l', g', g-g', t)ds'dl'dg' \end{aligned} \tag{2.5}$$

$$\begin{aligned} \Re_{agg}^{depletion} = F(s,l,g,t)\int_{s_{nuc}}^{s-s_{nuc}} \int_{0}^{l_{max}} \int_{0}^{g_{max}} \beta(s', s-s', l', l-l', g', g-g') \\ \times F(s',l',g',t)ds'dl'dg' \end{aligned} \tag{2.6}$$

where, s_{nuc} is the solid volume of nuclei, $\beta(s', s-s', l', l-l', g', g-g')$ is the size-dependent aggregation kernel that signifies the rate constant for aggregation of two granules of internal coordinates (s', l', g') and $(s-s', l-l', g-g')$. Although many aggregation kernels can be found in the literature,[33, 34, 35] we have implemented in our model, the empirical kernel proposed by Madec et al[36] :

$$\beta = \beta_0(V+V')\left((LC+LC')^{\alpha}\left(100 - \frac{LC+LC'}{2}\right)^{\delta}\right)^{\alpha}, \tag{2.7}$$

where

$$LC \text{ (Liquid binder Content)} = \frac{volume\ of\ liquid}{volume\ of\ agglomerate} \times 100. \tag{2.8}$$

and V_1 and V_2 represent the particle volume of the two particles aggregating. β_0, α, and δ are adjustable parameters, whose values can be found in Table 5.4.

2.1.2 Breakage

The $\Re_{breakage}$ term in the RHS comprises a breakage kernel and a breakage function. The phenomenon itself involves the disintegration of a larger particle into two or more smaller fragments and is mainly governed by attrition and impact. Considering this, the breakage term can then be divided into its corresponding birth and death terms as:

$$\Re_{break}(s,l,g) = \Re_{break}^{form} - \Re_{break}^{dep}, \tag{2.9}$$

such that the birth and death terms can be explained using equations (2.10) and (2.11)

$$\Re_{break}^{form} = \int_0^{s_{max}} \int_0^{l_{max}} \int_0^{g_{max}} K_{break}(s,l,g) b(s', s-s', l', l-l', g', g-g') \times F(s', l', g', t) ds' dl' dg' \tag{2.10}$$

$$\Re_{break}^{dep} = K_{break}(s,l,g) F(s,l,g,t). \tag{2.11}$$

The breakage function, $b(s', s-s', l', l-l', g', g-g')$ in Equation (2.10) has been obtained from literature.[37] Since one particle breaks down to form two smaller particles, there is an overall increase in the number of particles. Kernels for breakage can readily be found in the literature.[38,6] In this work we have used a previously proposed kernel[39] to address breakage. For specifics regarding the consolidation and re-wetting terms the reader is directed to subsections 2.2.3 and 2.2.4 respectively.

2.2 A four dimensional model for multicomponent granulation

In a pharmaceutical granulation process, the active pharmaceutical ingredient (API) is typically granulated with one or more excipients .[40] Modeling such a multicomponent granulation process would require the introduction of a second solid component, causing granule composition to become a dominant characteristic. Several attempts have been made in recent years to understand and subsequently model such a process.[40,41,42] With addition of more components, inhomogeneity in granule composition becomes a serious issue as it can reduce the uniformity in the final dosage form.[30,40] For this reason, a model that assumes homogeneous composition is inadequate. A fourth dimension must be added to the 3-D population balance model to completely account for granule composition.[30] For each additional solid component used, a new dimension must be added to the population balance model. This inevitably increases computational burden during simulation leading to even longer run times, which further stresses the need for parallelization. For the final case study, a four dimensional model has been parallelized to run across eight cores on two machines. The 4-D population balance was developed by modifying Equation 2.3:

$$\begin{aligned}\frac{\partial}{\partial t}F(s_1,s_2,l,g,t)+\frac{\partial}{\partial g}\left[F(s_1,s_2,l,g,t)\frac{dg}{dt}\right]+\frac{\partial}{\partial s}\left[F(s_1,s_2,l,g,t)\frac{ds}{dt}\right]\\+\frac{\partial}{\partial l}\left[F(s_1,s_2,l,g,t)\frac{dl}{dt}\right]=\Re_{aggregation}+\Re_{breakage}+\Re_{nucleation}\end{aligned}\tag{2.12}$$

where s_1 and s_2 represent the volumes of the two solid components in each granule.

2.2.1 Aggregation

The aggregation kernel is also modified as shown below:

$$\Re_{agg}(s_1,s_2,l,g,t)=\Re_{agg}^{form}(s_1,s_2,l,g,t)-\Re_{agg}^{dep}(s_1,s_2,l,g,t)\tag{2.13}$$

$$\begin{aligned}\Re_{agg}^{form}(s_1,s_2,l,g,t)=\frac{1}{2}\int_0^{s_1}\int_0^{s_2}\int_0^{l}\int_0^{g}\beta(s_1-s_1',s_2-s_2',l-l',g-g',s_1',s_2',l',g')\\\times F(s_1-s_1',s_2-s_2',l-l',g-g',t)F(s_1',s_2',l',g',t)\,\mathrm{d}g'\,\mathrm{d}l'\,\mathrm{d}s_2'\,\mathrm{d}s_1'\end{aligned}\tag{2.14}$$

$$\begin{aligned}\Re_{agg}^{dep}(s_1,s_2,l,g,t)=F(s_1,s_2,l,g,t)\\\times\int_0^{\infty}\int_0^{\infty}\int_0^{\infty}\int_0^{\infty}\beta(s_1,s_2,l,g,s_1',s_2',l',g')F(s_1',s_2',l',g',t)\,\mathrm{d}g\,\mathrm{d}l\,\mathrm{d}s_2\,\mathrm{d}s_1\end{aligned}\tag{2.15}$$

For aggregation, the kernel proposed by Madec et al.[36] was incorporated as it accounts for liquid binder content and granule size, both of which affect aggregation rate:

$$\beta(s_1, s_2, l, g, s_1', s_2', l', g') = \beta_0(V + V')\left((LC + LC')^\alpha \left(100 - \frac{LC + LC'}{2}\right)^\delta\right)^\alpha \tag{2.16}$$

where the two colliding particles are described by (s_1, s_2, l, g) and (s_1', s_2', l', g'). V and LC represent the total volume and fractional liquid binder content of the particles, given by Equations 2.17 and 2.18. β_0, α, and δ are adjustable parameters, whose values can be found in Table 5.5.

$$V(s_1, s_2, l, g) = s_1 + s_2 + l + g \tag{2.17}$$

$$LC(s_1, s_2, l, g) = \frac{l}{s_1 + s_2 + l + g} \times 100 \tag{2.18}$$

2.2.2 Breakage

Breakage occurs when particles disintegrate into two or more fragments due to impact and attrition. The breakage rate consists of the depletion of larger granules ($\Re_{break}^{dep}$) and the formation of smaller particles ($\Re_{break}^{form}$), as shown in Equations 2.19-2.21.

$$\Re_{break}(s_1, s_2, l, g, t) = \Re_{break}^{form}(s_1, s_2, l, g, t) - \Re_{break}^{dep}(s_1, s_2, l, g, t) \tag{2.19}$$

$$\Re_{break}^{form}(s_1, s_2, l, g, t) = \int_0^\infty\int_0^\infty\int_0^\infty\int_0^\infty K_{break}(s_1', s_2', l', g')b(s_1', s_2', l', g', s_1, s_2, l, g)F(s_1', s_2', l', g', t)\,\mathrm{d}g'\,\mathrm{d}l'\,\mathrm{d}s_2'\,\mathrm{d}s_1' \tag{2.20}$$

$$\Re_{break}^{dep}(s_1, s_2, l, g, t) = K_{break}(s_1, s_2, l, g)F(s_1, s_2, l, g, t) \tag{2.21}$$

The breakage kernel $K_{break}(s_1', s_2', l', g')$ describes the rate at which particles break, and the probability distribution function $b(s_1', s_2', l', g', s_1, s_2, l, g)$ determines the properties of the daughter particles. The fragments of a broken particle can vary in volume but must be smaller than the parent particle. For the purposes of this study, a uniform probability distribution was assumed for all possible daughter particle bins, represented by

$$b(s_1', s_2', l', g', s_1, s_2, l, g) = \frac{1}{(h-1)(i-1)(j-1)(k-1)} \tag{2.22}$$

$\forall\ (s_1, s_2, l, g) < (s_1', s_2', l', g')$, where h, i, j, and k are the first and second solid component, liquid, and gas bin numbers of the parent particles, respectively. According to this distribution, a parent particle is equally likely to break into fragments that can be described by each smaller bin. A semi-empirical breakage kernel was used in this study, as given by

$$K_{break}(s_1, s_2, l, g) = \frac{P_1 G_{shear}(D(s_1, s_2, l, g))^{P_2}}{2}, \tag{2.23}$$

where G is the shear rate, $D(s_1, s_2, l, g)$ is the particle diameter, and P_1 and P_2 are adjustable parameters.[39] The values for G_{shear}, P_1 and P_2 are included in Table 5.4.

2.2.3 Consolidation

Consolidation, a negative growth process representing the compacting of granules due to the escape of air from the pores, has been modeled using an empirical expression proposed by Verkoejin et al.[31] It can be given as

$$\frac{d\epsilon}{dt} = -c(\epsilon - \epsilon_{min}), \tag{2.24}$$

$$\frac{dg}{dt} = \frac{c(s+l+g)(1-\epsilon_{min})}{s} \times [l - \frac{\epsilon_{min} s}{1-\epsilon_{min}} + g] \tag{2.25}$$

where, the porosity ϵ for the 3-D case is

$$\epsilon = \frac{l+g}{s+l+g} \tag{2.26}$$

Here ϵ_{min} is the minimum porosity of the granules and c is the compaction rate constant. The same equations can readily be extended for the 4-D case, by re-defining the solid term, s, which will now be equal to the sum of individual solid fractions of each component, $s_1 + s_2$.

2.2.4 Drying/Rewetting

Liquid binder is added to the granulating system to induce the formation of particle aggregates. Drying/rewetting is associated with the change in the amount of liquid in the granulation system due to addition of more liquid or removal due to evaporation. The liquid rate can be obtained from mass balance as

$$\frac{dL}{dt} = \frac{\dot{m}_{spray}(1 - c_{binder}) - \dot{m}_{evap}}{m_{solid}}, \tag{2.27}$$

where,

$$m_{solid} = m_{solidfraction} + \dot{m}_{spray} c_{binder} \Delta t, \tag{2.28}$$

In the equations above, $\dot{m}_{spray}$ is the binder spray rate, c_{binder} is the concentration of solid binder in the slurry added, $\dot{m}_{evap}$ is the rate of liquid being evaporated (in this work $\dot{m}_{evap} = 0$, for sake of simplicity), $m_{solid\ fraction}$ is the volume of solid for the particles in each bin and L is the liquid content. Due to liquid addition, the liquid content of each particle changes from x_{liquid} to $x_{liquid} + \delta x_{liquid}$ which cannot be readily represented by the values of liquid volume on the grid. Thus, a fraction is incorporated, which distributes the new volume of liquid contained in the particle into the two adjacent grids, such that the liquid volume can be conserved. The fraction can be written as

$$fraction(j) = \frac{X - x(j)}{x(j+1) - x(j)} \tag{2.29}$$

where, $X = x_{liquid} + \delta x_{liquid}$, x(j) is the representative liquid volume in the j^{th} grid and x(j+1) is the representative liquid volume in the $j + 1^{th}$ grid.

Chapter 3

Parallel, distributed and GPU computing

Moores Law predicts that processing power of a CPU will double every 18 months,[43]a law shown to be reliable only for the duration of the 1990s primarily due to ever-increasing transistor counts per unit area with each new processor generation. By 2003, this aggressive rise in clock speed began to show signs of slowing down. To double the clock speed, distance traversed by an electrical signal per clock cycle has to be be cut by half, which in turn requires further reduction in size of transistors. To achieve this, manufacturers would have to place billions of transistors in proximity to each other on a single die. Such an arrangement will inevitably result in the generation of excessive heat leading to significant power leakages.[44] To overcome this, chip manufacturers are now fabricating CPUs with more (albeit lower clocked) cores instead of faster ones. A single processor can comprise many such cores, with each core capable of executing an instruction set. Although a 'core' generally refers to the physical component providing parallelism, it can also mean a thread (a piece of software), a processor or even a machine (on a network) executing a stream of instructions depending on the manufacturer/corporation.[45] For instance, the Intel Core i7-2600K processor has four physical cores, each with two threads raising the total number of (logical) cores to eight.[46] With an increased number of cores the burden falls on the programmer to optimally parallelize the code to achieve maximum speedup. The fundamental concept behind *parallel programming* involves splitting up a large problem into n smaller tasks that can be handled by n cores/processors to provide a peak speedup of n times over just one core/processor. However, attaining such gains in runtime performance are virtually impossible, simply because the time needed for data transfer and synchronization to, from and between cores eventually exceed any time savings from speedup.[3] Depending on the hardware architecture of a parallel computer, and implicitly, the level of communication required, several parallel programming models have been established, an elucidation of which can be found in the literature .[47,48] The most widely used approaches are task parallelism, data parallelism and the distributed memory/message passing model[49,50] :

- Task parallelism is achieved by assigning each task (or sub-task) to a unique core or thread (hence the name 'threads model') and finally splitting or combining the data stream at the end. Implementation: POSIX threads, OpenMP
- Data parallelism involves dividing a large amount of data into sections across cores, each of which are then operated upon by the same task within each core. Implementation: Fortran 90 and 95
- In Message Passing, each sub-task has its own local memory on the core and exchanges data between cores through messages. Programmer must explicitly determine the level of parallelism. Implementation: Message Passing Interface (MPI)

3.1 Multi-core CPU computing with the Parallel Computing Toolbox

Based on Flynn's taxonomy,[51] computing architecture can be classified as: single instruction, single data (SISD); multiple instruction, single data (MISD); single instruction, multiple data (SIMD); or multiple instruction, multiple data (MIMD) systems. The current generation of Intel processors like the Core i7 fall in the MIMD category but utilize SISD (Single Instruction, Single Data) processing units at the lowest level.[52] There are three common approaches toward implementing parallelism on these system: SIMD (SSE) instructions operating on multiple data sets in parallel with a single instruction stream; Simultaneous multi-threading (SMT), popularly called 'Hyperthreading'; or as is now generally preferred, through custom libraries or 'toolboxes' like MATLAB's Parallel Computing Toolbox (PCT). PCT allows the programmer to take advantage of multi-core processors and many-core devices like Graphics Processing

Units (GPUs) by providing high-level abstractions such as parallel `for`-loops with the `parfor` construct, specialized arrays (`distributed` and `codistributed` arrays), `SPMD` (Single Program Multiple Data) blocks, and frequently used numerical algorithms like `fft` that have already been parallelized.[53] This allows the programmer to focus on optimizing the algorithm and not on micro-managing parallel communication between cores, functions taken care of by MATLAB behind the scenes. Depending on the application, the programmer can choose one of these abstractions to parallelize his code to run on multi-core systems. By default, a task (a set of instructions) executed in MATLAB is handled by a master instance of MATLAB called the *client* instance. When the command `matlabpool open` is issued, a specified number of headless MATLAB instances (instances without an output display) called *workers* or *labs* are initiated on different cores and run as separate system processes. Communication to, from and between these workers are handled by the *client* instance of MATLAB. These workers are executed on cores, but their number need not correspond to the number of cores present on a device.

In addition to an implicit low-level multi-threading that is pre-built into MATLAB, there are explicit methods of parallelism available to the developer as well,[54] with the `parfor` keyword being the easiest to implement in an existing code requiring little modification. Substantial speedup can be achieved by just replacing (preferably) the outermost `for` with `parfor`. The job of distributing iterations and collecting end results are handled by MATLAB without any requirement for additional commands from the programmer. A major caveat of using this construct is that it assumes the task at hand to be 'embarrassingly parallel' in nature. That is, the task may be divided into a number mutually exclusive sub-tasks, which can then be executed independently of each other (i.e in any order) on separate cores.[3] In practice, it was found that implementing `parfor` - efficiently - in a program is easier said than done. This is mostly because any gain in speedup is soon lost when the number of labs exceeds the number of cores, since communication overhead is always higher between threads than for cores.[55] Another approach to explicit parallelism is by using SPMD blocks created with the help of the `SPMD` keyword. The SPMD (Single Program Multiple Data) approach belongs to a class of parallel programming models that utilize a combination of task, data and message passing paradigms to realize parallelism. Each MATLAB worker is assigned a copy of the same instruction set ('program') which then operate on different data arrays or different sections of the same data array; hence the term , 'Single Program Multiple Data'. Furthermore, if data exchange and synchronization between the workers is desired, functions based Message Passing Interface (MPI) commands [56] like `LabSend()` and `LabReceive()` can be applied to send or receive data respectively from a specified worker. To implement any of the constructs described here, a `matlabpool open` command must first be issued beforehand in order for the client session to establish a connection with available workers. For further information on PCT constructs and their correct implementation please refer the appropriate section of the MATLAB manual.[53]

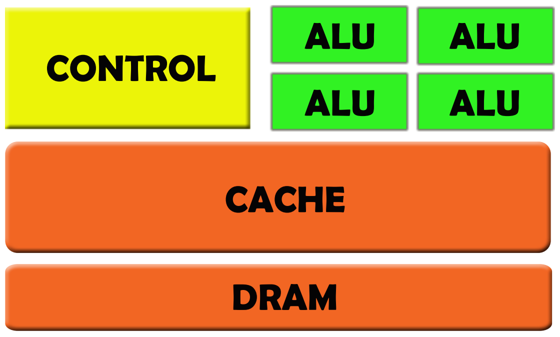
3.2 Distributed computing with PCT + MDCS

Once a program has been optimized for parallel execution on multiple cores on a single CPU or across multiple CPUs on a single machine, the next stage involves scaling out to a computer cluster, an approach known as 'farming'.[2] Each machine or a *node* in the network is linked to other nodes either via wired or wireless connections and are capable of communicating with each other. There are essentially two frameworks that describe algorithms for distributed systems: the shared memory and message-passing models. For the purposes of this paper, the message passing model will not be discussed but further information can be found in the literature.[57] In the shared memory - or more accurately, the *Distributed Shared Memory* (DSM) - structure it is assumed that CPUs of all nodes have access to a common memory where variables can be read and modified. There is no explicit message-passing between the nodes but exchange of information can be achieved by reading from/ writing data to this common memory. The computations themselves are performed in a tightly-coupled concurrent manner and is fundamentally a scaled-up version of parallel execution on a multi-core processor residing in a single machine.[58] The total memory in a DSM is not physically coalesced but distributed across local memory modules of every node, and together they form the global address space. While MATLAB generally manages all communication between nodes, there are two factors that must be kept in mind while writing code based on the DSM approach:[59] proper distribution of shared data throughout the system to minimize access latency; maintaining a coherent view of shared data across nodes while minimizing overhead associated with coherence management. Beyond a single machine or for one with more than 12 workers, MATLAB requires the Matlab Distributed Computing Server (MDCS) to be installed in order to harness more workers. Regardless of the underlying hardware structure, be it a multicore desktop or a computer cluster, all parallel constructs in the PCT will function in the same manner with the MDCS package.[53]

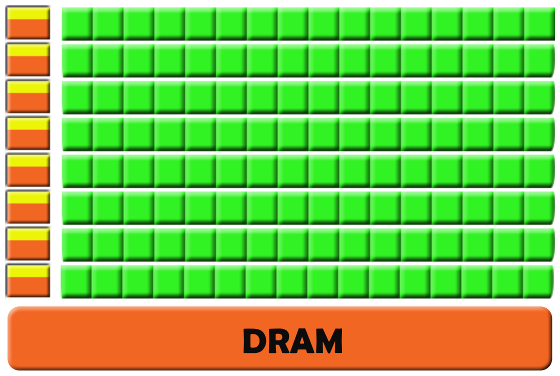
As distributed computing gains acceptance, corresponding challenges in scheduling computational tasks (or more appropriately *jobs*) to cluster environments also increase significantly. Such systems require job managing tools to ensure efficient and optimized resource allocation at the individual nodes to improve overall performance. With MDCS, communication to, from and between workers are managed by a software handler termed the *scheduler* running on top of the host operating system. In addition to the in-built 'job manager', MATLAB also supports third-party schedulers like PBS Pro and Windows HPC Server 2008. The primary function of the job manager is to control the job queue, distribute job tasks to workers or labs for execution, and maintain job results. Further information on configuring and submitting jobs through the job manager may be found in PCT's user guide.[53] Although MATLAB manages work flow and communication between client and workers, concurrent read/write access to shared data in a DSM environment can occur, resulting in a serious problem called 'false sharing'. Data transfer and reading/writing page table entries necessitates expensive system calls which increases the risk of false sharing.[60] Despite it being a complication only at page-level granularity, the programmer must keep this in mind while writing an application and endeavor to incorporate coarse-grain sharing and synchronization constructs in the parallel algorithm to ensure ordered data access. Another caveat of executing shared memory jobs on a distributed system is that they do not scale well with shared and/or updated data.[61] To run such an application in a truly scalable manner, the programmer must manually rework the code to partition both tasks and data across multiple nodes. This process revolves around choosing between complete data consistency or total scalability while simultaneously ensuring balanced loads across workers, rendering it a challenging task in and of itself. Therefore, it can be concluded that code optimized for tightly-coupled parallel execution on local workers may not be optimized for execution on workers in a distributed system, and vice versa.

3.3 GPU Architecture and parallel programming

For a comparison of hardware architecture layouts of both the CPU and GPU platforms, refer figure 3.1. The GPU is an excellent example of the SIMD design paradigm. A GPU is organized as an array of many cores, or 'streaming multiprocessors'(SMs), as NVIDIA describes them. Each SM has a certain number of ALUs (Arithmetic and Logic Units) called streaming processors (SPs) which share a common control logic and instruction cache (see Figure 3.1(a)). While the CPU design paradigm boasts excellent performance in sequential operations, presence of a complex control logic and large cache memory limit the maximum speed achievable in gigaflops.[62] The GPU control logic systems on the other hand are not as bulky, with the GPU themselves fabricated as relatively wide SIMD vector devices, increasing their parallel processing capacity. It is also on account of their architecture that GPUs are optimized for data-parallel calculations, unlike MIMD-based platforms like the Core-i7 CPU which are more suitable for task-parallel, data-parallel and message passing applications. The GPU card used in this investigation was a GeForce GTX 280 with 240 SPs each capable of a 1.3 GHz peak. Each SM has 1024 threads, bringing the total to 30,720 threads within a single GTX280 GPU.[63] To program these massively parallel architectures, NVIDIA developed the Compute Unified Device Architecture (CUDA) modeling platform, which was released in 2006, permitting high-level programmability within the C language environment[64] . CUDA was built upon the three key abstractions of: hierarchy of threaded groups, shared memories and barrier synchronization. CUDA, in conjunction with an Application Program Interface (API) greatly simplifies the process of GPU programming by transforming CPU code written using C, CUDA FORTRAN, OpenCL or DirectCompute, into GPU primitives. There are a number of custom libraries available to a GPU



(a) CPU architecture



(b) GPU architecture

Figure 3.1 Schematic representation of the contrast between typical CPU and GPU hardware architectures

programmer in addition to MATLAB's own built-in support via PCT, like GPUmat by the GP-you group and JACKET by the Accelereyes corporation. For this investigation, we decided to go with JACKET because of its extensive collection of GPU-ready functions and better performance when compared to the other products .[65,66] JACKET is a third-party MATLAB toolbox acting as a wrapper around CUDA transforming MATLAB functions into GPU functions at the basic level by converting CPU data structures into GPU types. This retains MATLAB's interpretive programming style while providing real-time, transparent access to the CUDA compiler[67] . Of all the available constructs, the `gfor` construct (similar to the PCT's *parfor*) was applied as it offered the easiest and most efficient way for parallelizing for-loops to run on the GPU. It executes for-loops in parallel by distributing the values of all loop iterations across GPU cores and subsequently executing calculations on each core in a single pass, resulting in considerable speedup. It must be kept in mind that for both the CPU and GPU, ideal parallelism is attained only if a task can be divided into a number mutually exclusive sub-tasks, capable of being executed independently of each other on separate cores - i.e. only if it is embarrassingly parallel in nature. In reality, most problems lie somewhere between this extreme and the ‘annoyingly sequential’ extreme.

Chapter 4

Model development and parallel programming

4.1 Building the multi-dimensional granulation model and its numerical solution

A modern view of the granulation process has been described previously[68] as a combination of three dominant mechanisms: (1) wetting and nucleation, (2) consolidation and aggregation and (3) breakage and attrition. These stages need not occur sequentially, but due to variable shear force distribution in granulators (e.g. high-shear) during operation, simultaneous growth and breakage of granules is presumed. In general, the entire process may be viewed as combinations of coalescence and/or breakage phenomena. We assume a class II type of coalescence,[69] meaning the granules were of negligible elasticity during an initial collision, attributable to the granules' deformability and/or being physically confined by surrounding granules. Weak, deformable granules imply that granular growth falls in the *steady-growth* region, a regime wherein coalescence is predominant over layering or nucleation.[70] Furthermore, Tu et al.[71] showed experimentally that steady-state growth could be observed under certain operating conditions (liquid:solid = 110:150 and impeller speed = 300 to 600 rpm) in high-shear granulators, with aggregation and breakage being the primary mechanisms for granule growth. Serial codes based on 3-D and 4-D PBEs were developed from equations (2.3) and (2.12) respectively to simulate the granulation process with a few simplifications, keeping the aforementioned assumptions in mind:

- Of the *source terms*, only aggregation and breakage are considered, eliminating nucleation ($s_{nuc} = 0$)

- Of the *growth terms*, layering is neglected, keeping liquid re-wetting and consolidation
- For aggregation, an empirical kernel proposed by Madec et al.[36] is used

yielding the following PBE (equation 4.1) for the 3D case:

$$\frac{\partial}{\partial t}F(s,l,g,t) + \frac{\partial}{\partial g}\left[F(s,l,g,t)\frac{dg}{dt}\right] + \frac{\partial}{\partial l}\left[F(s,l,g,t)\frac{dl}{dt}\right] = \Re_{aggregation} + \Re_{breakage} \tag{4.1}$$

and for the 4D case:

$$\frac{\partial}{\partial t}F(s_1,s_2,l,g,t) + \frac{\partial}{\partial g}\left[F(s_1,s_2,l,g,t)\frac{dg}{dt}\right] + \frac{\partial}{\partial l}\left[F(s_1,s_2,l,g,t)\frac{dl}{dt}\right] = \Re_{aggregation} + \Re_{breakage} \tag{4.2}$$

4.1.1 The cell average method

To cover the granule size ranges observed in an industrial setting, a linear grid would require a very large number of bins, leading to longer run-times. In order to reduce computational burden and achieve reasonable simulation times, a non-linear grid was used to define the bins in a grid. To distribute particles uniformly into bins in a non-linear grid, the cell average technique[72,73] was implemented. It is an improvement over the fixed-pivot discretization technique (FPDM) and moving pivot discretization techniques,[74,75] both of which were proposed to aid in the allocation of particles to neighboring bins when daughter particles did not exactly fall on the pivot (representative point of a bin, defined as the mid-point between the upper and lower bounds of a bin) in a non-linear grid. However, both techniques over-predicted the distribution of particle properties, a drawback that was rectified by the cell-average technique. It distributes birthed particles falling on non-representative points to the nearest representative points, with a pre-defined fraction of the total particles produced sent to neighboring representative points. A local averaging for each cell is performed at each time step before distributing the resulting values to the adjoining representative points for a non-linear, arbitrary grid. Likewise, during breakage, a larger granule will fragment into two smaller particles. Of the two daughter particles, one may be considered to lie on the grid point (the representative point) itself while the other lies at a point on the bin which may or may not coincide with this representative grid point. Therefore, this latter particle needs to be reallocated, in pre-defined fractions, to neighboring bins. To this end, the properties of all particles aggregating or breaking in a bin is averaged, representing net birth in a bin due to aggregation and breakage. This net birth is distributed to adjacent bins when the birthed particle does not fall exactly on the grid point. The actual number of adjacent bins receiving the particle depends on the number of dimensions present. Therefore, for a three dimensional grid, particles would get distributed to eight surrounding bins. For the model developed herein, the re-allocation method involves preservation of two moments, namely the particle density and mass (or volume, since particle density is constant). This fractionation can cause multiple births to occur at a single grid point. These births are summed up in order to obtain the cell average birth which then replaces the the birth term in the PBE. On the other hand, no redistribution of particles is needed for a death term, as death is simply the disappearance of particles from a grid point. For an N-dimensional PBE, it can be deduced via mathematical induction that a particle undergoing breakage would get distributed into 2^N fractions across adjacent grid points. The 2^N fractions corresponding to the 2^N neighbouring bins can be obtained in the same way as the 2^3 fractions for a 3-dimensional population balance equation. The detailed procedure for implementing this cell average method may be found in publications by Chaudhury et al. (forthcoming)[76] for a 3-D PBM, and by Barrosso and Ramachandran[77] for a 4-D PBM.

4.1.2 Numerical solution of the PB model

Using a multidimensional population balance with an appropriate kernel ensures improved analysis and prediction of the granulation process. However, incorporating an efficient numerical technique to solve such integro-partial differential equations is a challenging task, the multiple time scales and dimensions involved complicating the solution process. Hence the need to develop robust models with efficient solution techniques for such frameworks. Our approach for obtaining a solution to such equations is based on a hierarchical two-tiered algorithm as proposed by Immanuel and Doyle.[78] This involves using the finite volume approach for discretization with respect to individual solid, liquid and gas volumes, followed by first-order Euler integration of the population balance over the domain of these sub-populations. Neglecting layering and nucleation, the PBE to be solved is given in Equation (4.1):

$$\frac{\partial}{\partial t}F(s,l,g,t)+\frac{\partial}{\partial g}\left[F(s,l,g,t)\frac{dg}{dt}\right]+\frac{\partial}{\partial l}\left[F(s,l,g,t)\frac{dl}{dt}\right] \\ =\Re_{aggregation}+\Re_{breakage} \tag{4.3}$$

Equation (4.1) can be expressed in the discrete form as shown below in Equation 4.4:

$$\frac{dF'_{i,j,k}}{dt}+\left(\frac{F'_{i,j,k}}{\triangle g_k}\frac{dg}{dt}\Big|_{g_k}-\frac{F'_{i,j,k+1}}{\triangle g_{k+1}}\frac{dg}{dt}\Big|_{g_{k+1}}\right)+\left(\frac{F'_{i,j,k}}{\triangle l_j}\frac{dl}{dt}\Big|_{l_j}-\frac{F'_{i,j+1,k}}{\triangle l_{j+1}}\frac{dl}{dt}\Big|_{l_{j+1}}\right) \\ =\Re_{agg}(s_i,l_j,g_k)+\Re_{break}(s_i,l_j,g_k) \tag{4.4}$$

Here $F'_{i,j,k}=\int_{s_i}^{s_{i+1}}\int_{l_j}^{l_{j+1}}\int_{g_k}^{g_{k+1}}F(s,l,g)\,ds\,dl\,dg$, s_i is the value of the solid volume at the upper end of the i^{th} bin along the solid volume axis, l_j is the value of the liquid volume at the upper end of the j^{th} bin along the liquid volume axis, g_k is the value of the gas volume at the upper end of the k^{th} bin along the gas volume axis $\triangle s_i$, $\triangle l_j$ and $\triangle g_k$ are the sizes of the i^{th}, j^{th} and k^{th} bin with respect to the solid, liquid and gas volume axis. Using this technique, the population balance equation, is reduced to a system of ordinary differential equations in terms of the rates of nucleation ($\Re_{nuc}(s_i,l_j,g_k)$), aggregation ($\Re_{agg}(s_i,l_j,g_k)$) and breakage ($\Re_{break}(s_i,l_j,g_k)$). Hence, the triple integral associated with the aggregation term may be evaluated by casting it into simpler addition and multiplication terms. This is followed by fractioning the resulting aggregation term into its neighbouring grids through the previously described cell-average technique. It is assumed that the parent particle will lie exactly on a grid point before breakage, with one of the daughter fragments staying on the grid point, post-breakage. The other fragment may or may not fall exactly on the volume grid point and is therefore reassigned to the adjoining bins via cell-averaging. The fragment that is assumed to lie exactly on the volume grid is dealt with independently of the other daughter fragment. The two resulting formation terms for each of the fragments are then summed up to obtain the overall birth terms due to breakage.

This approach is extended for application to the 4-D population balance model via inclusion of the second solid component term. The resulting four dimensional population balance equation was solved numerically as for the three dimensional case. The descriptive particle property parameters were again discretized into bins according to the granule volume of each solid, liquid, and gas components. Because the 4-D model is so computationally expensive, a smaller number of bins were necessary to solve this model in an acceptable amount of time. Furthermore a non-linear grid was initialized to define these bins, as shown in equations 4.5-4.8.

$$s_{1,h} = s_{1,1} \times 3^{h-1} \tag{4.5}$$

$$s_{2,i} = s_{2,1} \times 3^{i-1} \tag{4.6}$$

$$l_j = l_1 \times 3^{j-1} \tag{4.7}$$

$$g_k = g_1 \times 3^{k-1} \tag{4.8}$$

The indices h, i, j, and k represent the bin numbers in the four dimensions. In the (h, i, j, k) bin, the volume per particle of each component is given by $(s_{1,h}, s_{2,i}, l_j, g_k)$. The initial population density is distributed over pre-defined bins, usually the first one designated by $(1, 1, 1)$ for the 3-D case or $(1, 2, 1, 2)$ and $(2, 1, 1, 2)$ for the 4-D case. The integration was performed to track the population distribution over time, with a fixed time step. As with any explicit numerical solution to partial differential equations, numerical instability can occur if the selected time step is too large. The Courant-Friedrichs-Lewis (CFL) condition, shown in Equation 4.9, must be satisfied:.[13]

$$\frac{GR\Delta t}{\Delta s_1} + \frac{GR\Delta t}{\Delta s_2} + \frac{GR\Delta t}{\Delta l} + \frac{GR\Delta t}{\Delta g} = CFL < 1 \tag{4.9}$$

In this case, the growth rate, GR, is the rate of change in volume due to consolidation or liquid addition. This condition indicates that the time step must be less than the time required for particles to travel to adjacent grid points.

4.2 Programming strategy for parallel implementation

Once the serial version of the code has been debugged, partial *vectorization* of the code is carried out to ensure most calculations are performed as efficiently as possible. Vectorization or vector processing is based on Flynn's definition of vector processors[79] to mean a single instruction stream capable of operating on multiple data elements in parallel. In vectorized code, an operation is performed on all (or multiple) elements of the input variables in one statement i.e the operands are treated as single vectors. On the other hand in a non-vectorized code, operations are performed element-wise by treating each operand as a scalar, using loops to index each element of the array. After building a sequential version, the code is parallelized for execution on multiple workers. The procedure for parallel programming involves three basic steps: locating portions of the code that are most time-consuming with tools like the MATLAB profiler; applying one of the approaches for parallelism outlined previously(task, data, or message passing models) as appropriate; and finally optimizing the code for minimal variable transfer overhead. A flowchart representation of this strategy is given in Figure 4.1.

4.2.1 Prioritizing

The first step is to identify routines that take the most time to run and prioritize them to obtain most speedup. For this purpose, the MATLAB profiler is indispensable, allowing the programmer to profile the code and locate with precision those statements that are called the most, and time needed for their execution. Based on the profiler results, one can then carefully choose portions of the code to be rewritten to yield the best performance without sacrificing scalability. There are some calls which cannot be avoided, such as those invoked to start a pool of workers and synchronize them at intervals, overheads from built-in functions etc. These cannot be parallelized or circumvented

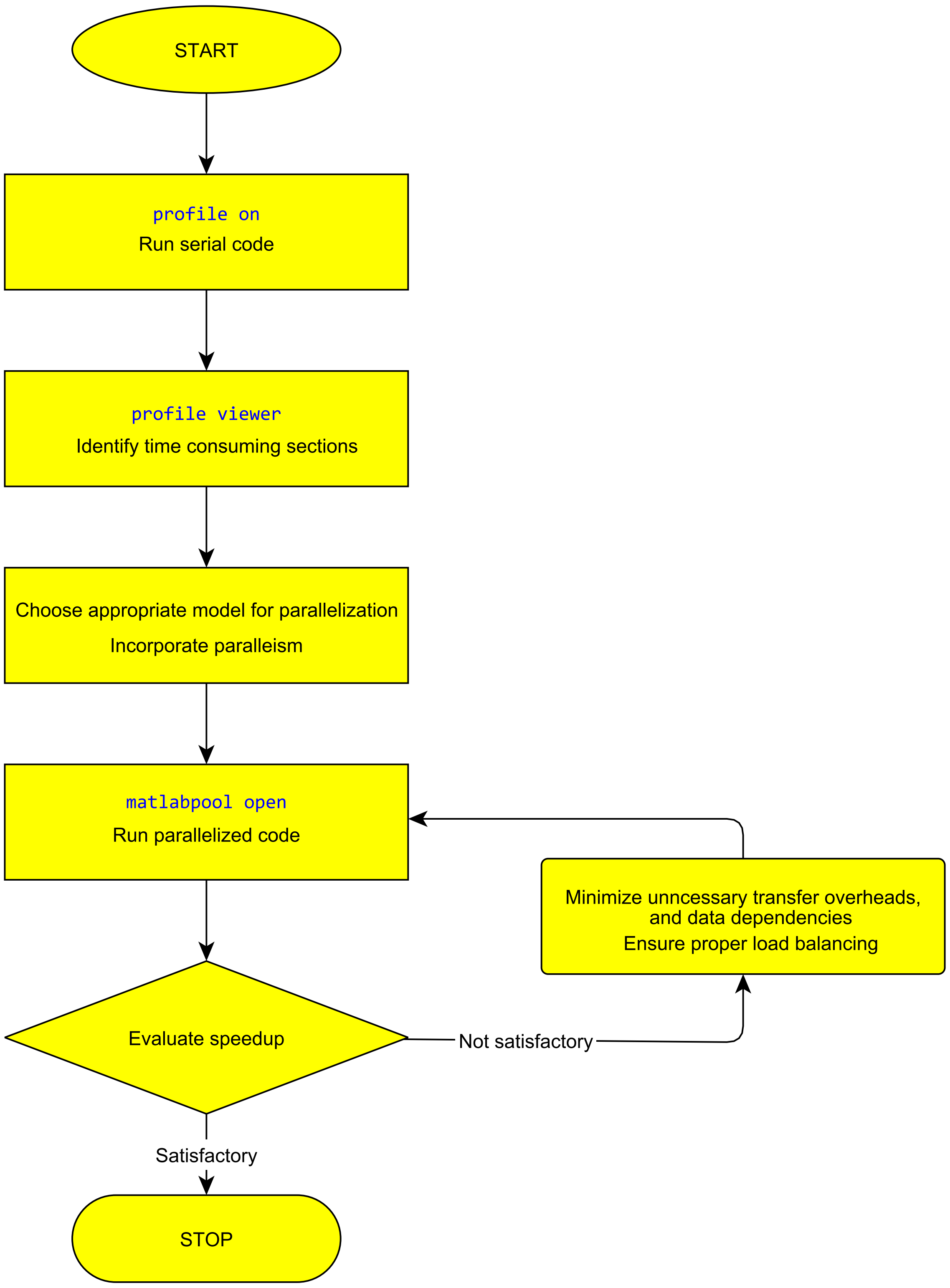

START
profile on
Run serial code
profile viewer
Identify time consuming sections
Choose appropriate model for parallelization
Incorporate paralleism
matlabpool open
Run parallelized code
Evaluate speedup
Not satisfactory
Minimize unncessary transfer overheads, and data dependencies
Ensure proper load balancing
Satisfactory
STOP

Figure 4.1 Flowchart depicting key steps in parallel programming

easily but their computational costs are usually one-time expenses and not too significant. As shown in the next section, the function computing for aggregation proved to be the most resource-intensive for all simulation cases, followed by the calculations to relocate the newly-birthed particle phase fractions into adjacent bins via cell-averaging in both 3D and 4D PB models. Aggregation and its associated cell average calculations are therefore the main computational bottlenecks in the PBM, rendering them prime candidates for parallelization. Their computational burden can be attributed to the presence of several 'nested `for`-loops', prominently those accounting for integral equations 2.5 and 2.6, which are performed element-wise and sequentially on a single CPU core. Consequently, broadening the range of each loop index causes individual iterations to run even slower. Increasing the number of bins in each grid, while raising the dimensionality of the system, also slows down code execution considerably. This is termed the *curse of dimensionality phenomenon.* Although it is preferred to use a higher grid size for a more accurate representation of the system, the aforementioned limitations curb the degree of flexibility available to a researcher trying to simulate the system. There is therefore much potential for speed-up in parallelizing these loops to run simultaneously on all cores/processors present on the cluster.

4.2.2 Choosing a parallel programming paradigm

The data parallel approach

Once potential sections have been identified for parallelization, the next step is to decide on a parallel programming paradigm most suited to the script and hardware. Implementing parallelism with respect to a CPU, GPU and cluster involved the SPMD paradigm outlined in the previous chapter to achieve data parallelism. Though the bulk of the PBM code is 'annoyingly sequential' in nature, it is less computationally intensive than the aggregation kernel, which is where the potential for parallelism exists. The aggregation kernel (assuming a three-dimensional form) typically comprises of 6 nested for-loops, with two sets of three loops each, to account for interactions between the s, l, and g fractions of two colliding particles in a bin. Since each MATLAB worker is designed to operate independently of each other with all communications handled by the client instance, the best approach is to decompose the index space adequately by a process known as loop-slicing.[80] The first step in the process is to identify loop axes (a range of loop index values) capable of functioning as indices for parallelism, followed by assigning these loop axes to available MATLAB workers, `numlabs`, (preferably equal to the number of cores on the parallel device) by means of `labindex` . `Numlabs` returns the number of workers open in a given `matlabpool` session, while `labindex` returns the currently executing worker's index. Loop orders may be switched for efficient memory access patterns and axes may be further sliced if the device memory is found to be insufficient for a given loop size. For parallel execution on a multi-core CPU, following the SPMD (a type of data-parallel) approach seemed most appropriate due to the embarrassingly parallel nature of the computations for aggregation and breakage. Moreover, it would scale well on a distributed system with shared memory (DSM), further justifying adoption of this particular mode of parallel programming. Following the procedure just described, execution of the aggregation kernel (4D) can be parallelized by slicing the outermost loop:

$$\begin{aligned}\Re_{agg}^{formation} = & \int_0^{size_1}\int_0^{l_{max}}\int_0^{g_{max}} form(s,l,g) + \int_{size_1+1}^{size_2}\int_0^{l_{max}}\int_0^{g_{max}} form(s,l,g) \\ & + \int_{size_2+1}^{size_3}\int_0^{l_{max}}\int_0^{g_{max}} form(s,l,g) .... + \int_{size_n+1}^{s_{max}}\int_0^{l_{max}}\int_0^{g_{max}} form(s,l,g)\end{aligned} \tag{4.10}$$

where,

$$size_n = \frac{s_{max}}{\texttt{numlabs}} \times \texttt{labindex} \tag{4.11}$$

and

$$\begin{aligned}form(s,l,g) = & \frac{1}{2} \times \beta(s', s-s', l', l-l', g', g-g') F(s', l', g', t) \\ & \times F(s', s-s', l', l-l', g', g-g', t)\end{aligned} \tag{4.12}$$

In MATLAB code, this slicing would translate as:

$$\texttt{for}\ i = \frac{(\texttt{labindex} - 1) \times (grid\ \ size)}{\texttt{numlabs}} + 1 : \frac{\texttt{labindex} \times (grid\ \ size)}{\texttt{numlabs}} \tag{4.13}$$

The parallel algorithm implemented for solving the PB model with cell averaging and breakage on a distributed system is shown in Figure 4.2. A simplified version same algorithm is used for parallelizing the 3-D code for a multi-core CPU. For parallel implementation on the GPU, the `gfor` functionality of the JACKET toolbox was utilized. JACKET's `gfor` employs an algorithm similar to the loop slicing technique to distribute sections of a for-loop on a GPU, so the programmer does not have to explicitly manage communication to, from and between workers. The source terms are encapsulated in function calls to enable faster and better access to kernels like aggregation, improving the overall parallel performance. Kernels are called from within the `gfor` loop at every time step. The `gfor` is preferably kept as the outermost loop, as it can parallelize all subsequent statements in a single pass, but due to the presence of 6 nested `for` loops, we decided to replace the fourth `for` with the `gfor` loop to minimize the number of kernel calls and thus reduce memory transfer overheads.

The task parallel approach

Besides data-parallelism, another, more straightforward *divide-and-conquer* approach involves task-parallelism, also known as the MIMD (Multiple Instruction Multiple data) paradigm. Implementations of task-parallelism are generally done through the fork-join model, described in Refianti et al,[81] which relies on multiple threads executing blocks of sequential code to achieve parallelism. Here, a multiprogramming style was adopted in order to easily achieve *coarse-grained* parallelism. This means partitioning the problem into fewer, but larger tasks able to execute independently of each other. These discrete sections of code are mapped onto different threads that execute asynchronously and independently of each other. Task scheduling is done at the time of compilation i.e. statically. A major shortcoming of this approach is the static nature of task distribution which leaves the granular complexity of task unbounded. A task with unbounded or

```
matlabpool open distributed_config nlabs
% define inputs and perform initial calculations
....
while time  < final_time do
      % send data to workers
      spmd (nlabs)
            % perform calculation on workers
            for   $\frac{(labindex-1)*grid\_size}{numlabs} + 1 : \frac{(labindex)*grid\_size}{numlabs}$
                  % calculate  $\mathfrak{R}_{agg}^{form}$  and  $\mathfrak{R}_{agg}^{dep}$
                  % calculate  $\mathfrak{R}_{break}^{form}$  and  $\mathfrak{R}_{break}^{dep}$
            end
            output = gplus(output,1) % global summation across workers
      end
      %gather results and send back to client
      %calculate output variables and update $F(s,l,g)$
end
```

Figure 4.2 Algorithm for distributed execution describing the loop-slicing technique in conjunction with the SPMD keyword.

variable task size means inefficient CPU usage, since every task runs for different periods of time depending on the size of the problem and consequently exit workers at different times.[82] Another limitation of task parallel algorithms is the restriction of maximum degree of parallelism achievable to the number of individual tasks. In contrast, data parallel algorithms can be readily scaled to (theoretically) any number of processors (Figure 4.3). Furthermore, due to the variability in run times, task parallelism requires micro-managing communication and synchronization to balance the computational load across processors.[83]

4.2.3 Parallel execution procedure and code optimization

At the start of the simulation, only the MATLAB client instance is actively processing code sequentially. On reaching an SPMD keyword, the code forks off function calls onto idle workers concurrently. With every worker active, execution of the allocated serial tasks now begins asynchronously. After all the workers have completed their respective tasks, the results are summed over all workers (with `gplus`) and the sum cast to one of them. Any result on a worker is of the `Composite` type, but can readily be cast back to regular CPU `single` type on the client MATLAB instance and subsequently re-joined. Execution on the GPU essentially follows the same protocol except that JACKET handles all communication between the processor and GPU device. After calculations, all `gsingle` and `gdouble` (GPU-class) variables are converted back to CPU-class variables before re-joining. The final step involves evaluating speedup obtained and fine-tuning the code for optimized performance if necessary. There is no standard protocol to follow while re-structuring parallel code to achieve optimal speedup. Although there are several reasons for poor parallel performance, it is typically due to the work handled per processor not being sufficient enough to outweigh the computational costs of concurrent processing. Therefore, it is the type and complexity of a problem that dictates the need for parallelization and the paradigm to be incorporated. As a rule of thumb, the programmer must strive to: minimize data transfer overheads; reduce data inter-dependencies; and finally, balance computational loads across all cores. This final step is critical to ensure robust performance and scalability. Furthermore, achieving near linear speedups requires significant tweaking of the original code, and sometimes having to micro-manage communications between the device memory and processors. To sum up, the procedure followed herein for parallelizing PBMs for distributed systems involved three steps: locating portions of the code that are most time-consuming with tools like MATLAB profiler; applying one of the aforementioned approaches for parallelism as appropriate; and lastly minimizing overheads associated with variable transfer, data dependencies and load balancing.

Chapter 5

Case studies: Results and discussion

5.1 Comparing CPU-`for`, GPU-`for` and GPU-`gfor` execution

Profiler analysis

The first set of simulations were conducted to compare the speed gains obtained by running an aggregation-only PB code, based on a simplified form of equation (4.1). To identify computational bottle-necks in the serial version of the code, MATLAB's profiler utility was employed and the results graphed as shown in Figure 5.1. From the figure it is apparent that aggregation, and more specifically, formation, is indeed the most computationally intensive section of the code consuming up to 73% of the total simulation time. This makes formation the statement most suited for incorporating data-parallelism. There are 6 nested `for` loops that account for aggregation, enabling the loop slicing technique described in the previous section to be readily implemented. JACKET's `gfor` construct is particularly advantageous in this regard in that it effectively assigns the same task to operate on different partitions of the shared array concurrently. The formation and depletion functions are consolidated into a single function and the fourth nested `for` loop was replaced with the `gfor` in order to minimize overheads due to excessive function calls. This paralleized code was tested on two platforms: first on a GPU and secondly, on a single CPU core. For the GPU, two parallel versions of this code were investigated: In one case, standard `for` loops were executed on the GPU over GPU-class variables (termed the 'gpu `for`' version); and the other, termed the `gfor` version, utilized JACKET's `gfor` constructs to loop over the same GPU-class variables. The CPU version was left un-parallelized, i.e. with regular `for` loops, to execute sequentially on a single MATLAB worker. Simulation was

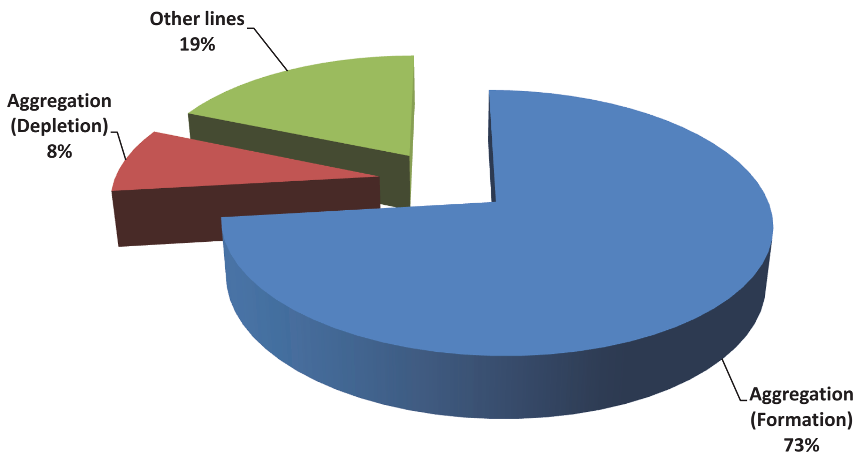


Figure 5.1 Piechart representation of MATLAB's profiler results for a serial version of the 3-D granulation code with only aggregation, run on a single worker.

carried out on a machine with a Core2Quad Q6600 processor (2.4 GHz clock, 4 cores, no threads), 4 GB of RAM (2 GB $\times$ 2 sticks) and an NVIDIA GeForce GTX 280 GPU (240 CUDA cores, 1296 MHz processor clock, 1 GB memory).

Numerical accuracy validation

Results from the simulation of each of these three cases were first validated by comparing bulk property plots of total number of particles vs time, total volume vs time and average diameter vs time after the final time step to verify uniformity. From the curves depicting temporal evolution of granule properties (Figure 5.2), it is clear that numerical accuracy of the computations was not compromised during execution either on the CPU or GPU as the curves in each plot coincide perfectly with one another. As expected, the total number distribution of particles (Figure 5.2(a)) decreases at a constant rate due to aggregation, wherein the collision (and therefore, depletion) of two particles lead to the formation of a new one by coalescence [11] . An analysis of the total volume plot, Figure 5.2(b) predictably reveals constant value lines considering the fact that total mass/volume is conserved in the system i.e. no particles are either added to or removed from the system during the process. The volume of a new, larger granule is equal to the sum of the volumes of the smaller coalescing particles that formed it. By extension, this is the reason why the average granule diameter plot, Figure 5.2(c) shows a proportional increase in size of granules over time.

Performance evaluation

To evaluate parallel performance on the GPU device, the time taken to simulate each case was plotted against grid size , followed by the performance ratio against grid size. These ratios were calculated as:

$$\text{Performance Ratio} = \frac{\text{single CPU time}}{\texttt{gfor}\ \text{time}} \tag{5.1}$$

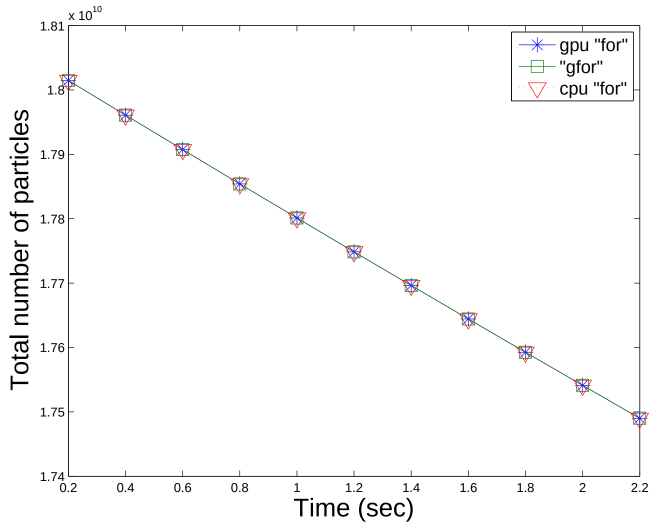
or

$$\text{Performance Ratio} = \frac{\text{GPU}\ \texttt{for}\ \text{time}}{\texttt{gfor}\ \text{time}} \tag{5.2}$$

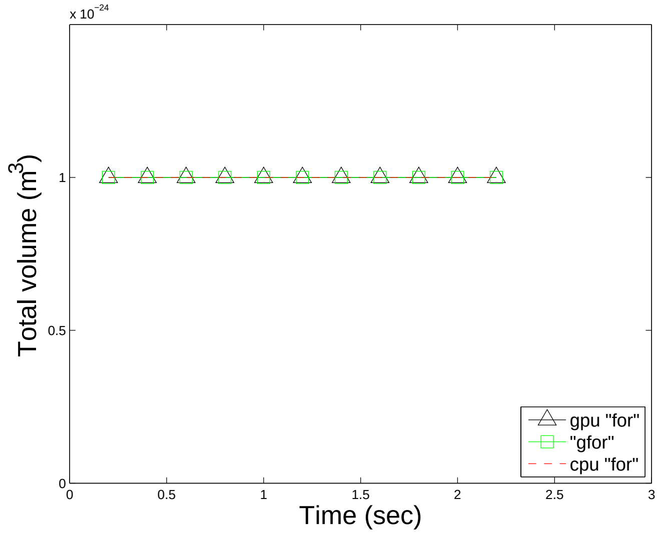
The simulation time vs grid size curves in figure 5.3 show the single-worker CPU version of the code to be much faster than its GPU counterparts, with the slowest of the set being the code with GPU `for` loops, followed by the `gfor` loop version. It must be noted that the GPU is a stand-alone device and does not share its memory with the host (CPU) or provide a means for combined virtual memory addressing. In other words, data will not be communicated automatically between the host and the device memories, but must be explicitly invoked. This results in severe memory transfer overheads each time a variable is copied to and from the GPU across the PCI-E bus,[84] which is why the GPU versions are drastically slower than their CPU counterparts. Furthermore, while the INTEL Core2Quad Q6600 CPU can achieve processor clock speeds of 2.4 GHz, the GPU core clocks in significantly lower at 1.3 GHz forcing the same computations to take longer to run on the GPU. As anticipated, the code with `gfor` ran faster than just `for` on the GPU owing to `gfor`'s inherent ability to schedule and control loop distribution. This speedup is readily discerned in figure 5.3(c), with the ratio calculated by equation 5.2.

Although these preliminary results seem to indicate that CPUs are better than

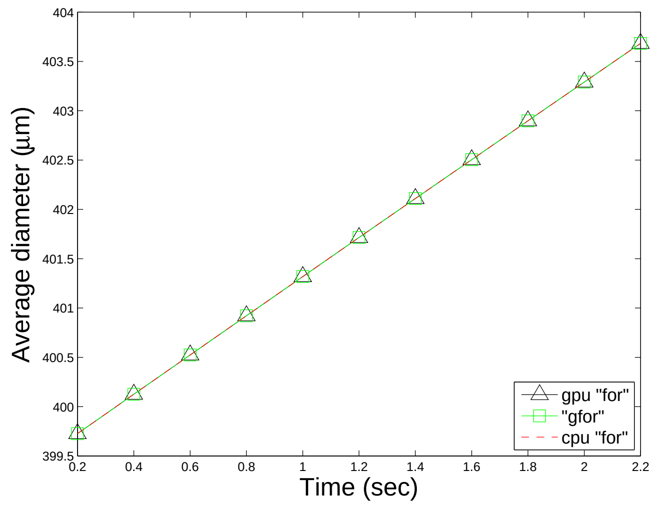
GPUs for this program, the trend quickly reverses as we increase the size of the grid (and implicitly, the resolution of the system) beyond 11, as suggested in figure 5.3(b). The steady increase in the ratio (equation 5.1) curve implies that the simulation time curves for `gfor` and CPU-`for` are converging and will eventually meet at some particular grid size, after which the GPU will perform significantly better than the CPU in a progressive manner. Beyond a grid size of 20 it became impractical to run the code for extensive periods of time and therefore further investigations were not carried out. The initial drop seen in the CPU `for` curve in (Figure 5.3(a)) and in the Figure 5.3(b) is an anomaly, shown to be reproducible even after initiating the simulation at various grid size values and is likely due to an initial spike in memory overhead during the 'warm-up' of the CPU prior to commencement of execution. In addition to the aforementioned hardware limitations of the GPU, JACKET's execution of a script in not transparent to the programmer, and therefore capabilities such as benchmarking, assigning tasks to specific thread blocks, and controlling memory access patterns is non-existent.



(a) Evolution of total number distribution of particles over time

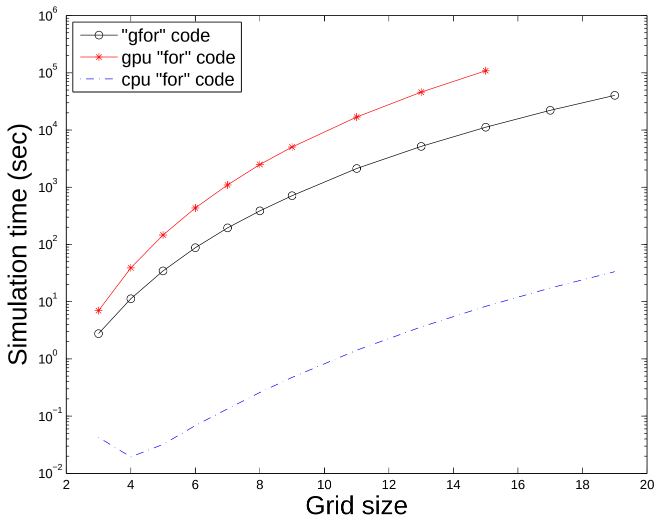


(b) Evolution of total volume of particles over time

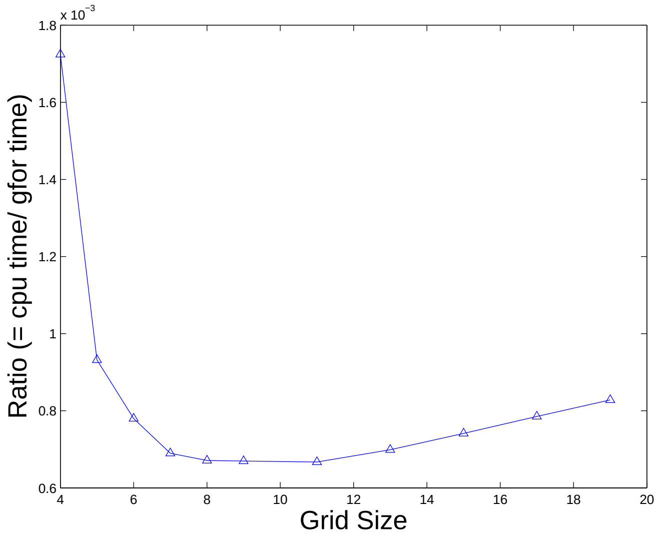


(c) Evolution of average diameter of a particle over time

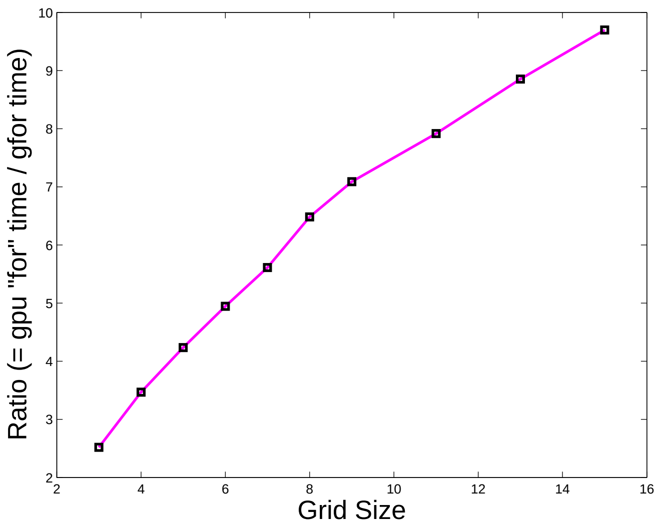
Figure 5.2 Comparison of temporal evolution of granule physical properties simulated using `gfor`, GPU-`for` and CPU-`for`



(a) Semi-log plot comparing simulation times of `gfor`, GPU `for` and CPU `for` versions



(b) Speedup ratio of `gfor` over CPU `for` version



(c) Speedup ratio of `gfor` over GPU `for` version

Figure 5.3 Comparison of simulation times and performance ratios of PBM code incorporating `gfor`, GPU-`for` and CPU-`for`

5.2 Comparing single CPU to SPMD execution

Profiler analysis

A comparison of the simulation times needed by the PB code to run on a single CPU thread sequentially and on multiple CPU threads was done, followed by plotting the speedup gained. Prior to execution, the code was 'streamlined' to efficiently search for and perform computations on relevant particle-containing bins in a grid, unlike the version employed in the previous section which looped over all bins irrespective of whether particles were present. This optimization was carried out to eliminate the time spent on unnecessary calculations, specifically with respect to empty bins. The GPU version could not be streamlined since our version of JACKET did not allow for conditional branching within `gfor` loops.[67] The serial version of this modified code was run on a single CPU worker, and an analysis of time consumption was carried out with the aid of MATLAB's profiler tool. A breakdown of the time spent on various calls is displayed in Figure 5.4. It is obvious once again that aggregation must be parallelized in order to see an improvement in performance. Both formation and depletion are are the main bottlenecks in the code, with formation being slightly more compute-intensive (51% and 47% respectively). Parallelism was attained with the loop-slicing technique described in section 3. The formation and depletion loops were sliced in accordance with the pool of MATLAB workers available (one, two, four, six and eight threads) to analyze the gain in speedup and effects of transfer overhead. The new streamlined code was run on an Intel Core i7-870 CPU (4 cores, 8 threads, 2.93 GHz clock speed) with 8 GB of RAM. To determine the most appropriate index range for loop discretization, different combinations of sliced formation and depletion loops were tested for efficiency of parallelization (refer Table 5.1). Although formation is the primary computational bottleneck requiring loop-slicing, initial test runs in conjunction with MATLAB's Profiler tool affirmed that it was also necessary for depletion to execute on at least one thread for the gain in speedup to outweigh memory transfer overhead. Consequently, certain combinations based on grid size and number of workers were discarded, with only pertinent ones being retained. From within these combinations, the ones yielding

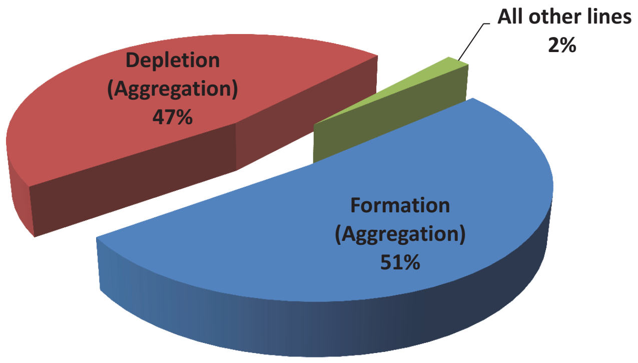


Figure 5.4 Piechart representation of MATLAB's profiler results for the streamlined version of the 3-D granulation code with only aggregation, run on a single worker.

the lowest simulation times for a grid size of 36 were chosen from each worker pool class for comparative analysis: 0 formation, 0 depletion (1 thread); 1 formation, 1 depletion (2 threads); 3 formation, 1 depletion (4 threads); and 6 formation, 2 depletion (8 threads).

Numerical accuracy validation

As done previously, the plots for granule physical properties, Figures 5.5, were examined to ensure validity of the data and numerical precision of the results. The total number distribution of particles (Figure 5.5(a)) decreases at a constant rate due to particles coalescing. Predictably, the total volume plot, Figure 5.5(b) shows a constant value

Table 5.1 *Combinations for loop slicing*

Number of workers (worker pool)	Number of times sliced	
	Formation	Depletion
1	0	0
2	1	1
4	2	2
	3	1
	4	0
6	3	3
	4	2
	5	1
8	6	2

considering total mass/volume is conserved i.e. no particles are either added to or removed from the system during the process. The volume of a new, larger granule is equal to the sum of the volumes of the smaller coalescing particles that formed it. And because particles coalesce during aggregation, the average granule diameter plot in Figure 5.5(c) shows a proportional increase in size of granules over time. Furthermore, all plots coincided perfectly for every parallel simulation case, affirming the retention of their numerical accuracy.

Performance evaluation

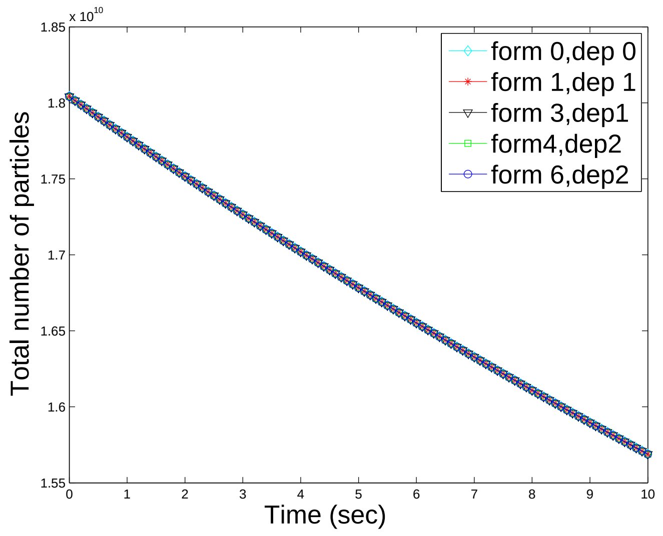
Having confirmed that numerical precision was not compromised, the simulation times, the parallel speedup and efficiency curves were plotted for the five worker pool classes selected (Figures 5.6a-c). The speedup factor and parallel efficiency were calculated as given in Wilkinson et al.[3]

$$\text{Speedup, } S(n) = \frac{\text{Execution time on a single worker}}{\text{Execution time on } n \text{ workers}} \tag{5.3}$$

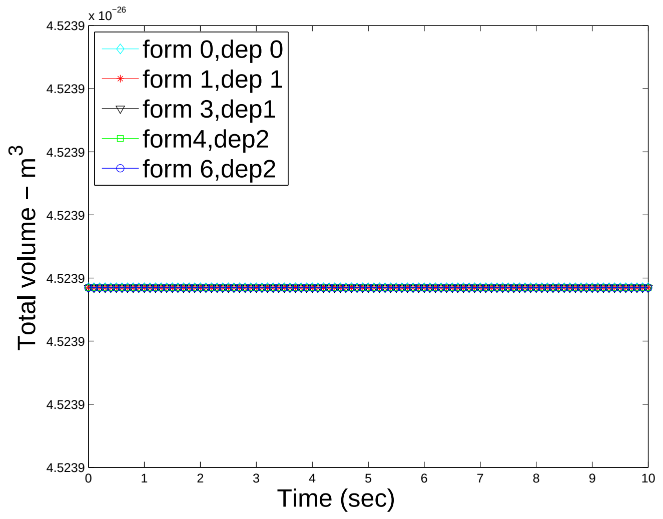
$$\text{Parallel Efficiency, } E_n = \frac{S(n)}{n} \times 100 \tag{5.4}$$

Simply put, the speedup factor directly quantifies the gain in performance of multi-processor system over a single processor one. As observed in figure 5.6(b) (b), the maximum speedup achieved with 8 workers was 2.2 times, leading to an average per worker efficiency of 27.35%. Parallel efficiency is a measure of computational resource usage, with lower values implying lower utilization and higher values implying better utilization on average. Although the speedup achieved for a grid size of 36 was marginal, it was theorized that an increase in the problem size would improve not only speedup but also parallel efficiency. As expected, an increase in grid size to 60 positively affected both the speedup and efficiency of parallel execution as seen in figure 5.6(c). Furthermore, it was also observed that the most efficient way of parallelization for 6 cores was by splitting formation 5 times and depletion 1 time, as opposed to the previous strategy of splitting formation 4 times and depletion twice. Since depletion is much less computationally intensive than formation and only becomes challenging at higher grid sizes, this finding is in line with our expectation that each worker has to have sufficient work for parallelism to pay off. That is, for a fixed pool of workers, an increase in problem (grid) size will mean improved speedup. This will also explain the drop in efficiency as well as speedup from 4 to 6 workers (i.e. with formation sliced 4 and depletion 3 times, figure 5.6(b), 5.6(a)).

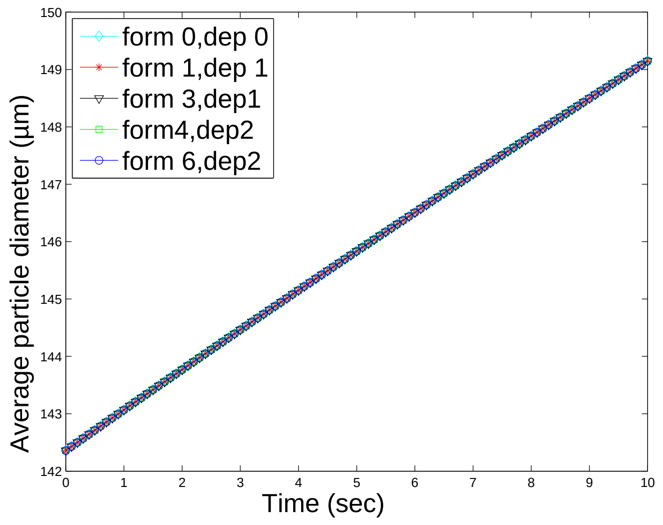
Currently, we have restricted ourselves to a grid size of 60 due to MATLAB's limit on the maximum possible array size, which is proportional to available system RAM. Of the three main factors that might have impacted the efficiency of our parallel algorithm, load balancing and data dependency were ruled out as the `for`-loops were split evenly across workers with each loop capable of independent execution on a worker. Thus the only possible reason could be overheads resulting from communication between workers. These overheads are generally the result of: computational costs of cache coherence; memory conflicts inherent to a shared-memory multiprocessing architecture like the INTEL Core i7;[85] and memory conflicts between operating system services.[86] Moreover, since MATLAB looks to the Operating System to open a pool of workers, it does not guarantee proper assignment of each worker to a single physical core/thread, which would result in exaggerated overheads from different worker instances trying to communicate with (or waiting for) other instances on the same thread. It must be kept in mind that there are always statements in code that cannot be parallelized, which limits the maximum speedup theoretically attainable. It is also interesting to note that since there are only 4 physical cores which correspond to 4 processing units, the speedup ratios achieved for the two grid sizes (2.2 and 2.65 times) can be considered effectively out of a 4 times ideal speedup, meaning an improvement of about 66%.



(a) Evolution of total number distribution of particles over time

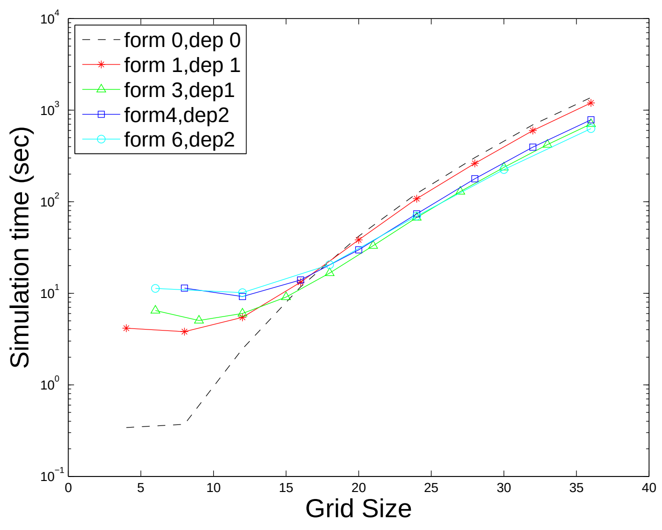


(b) Evolution of total volume of particles over time

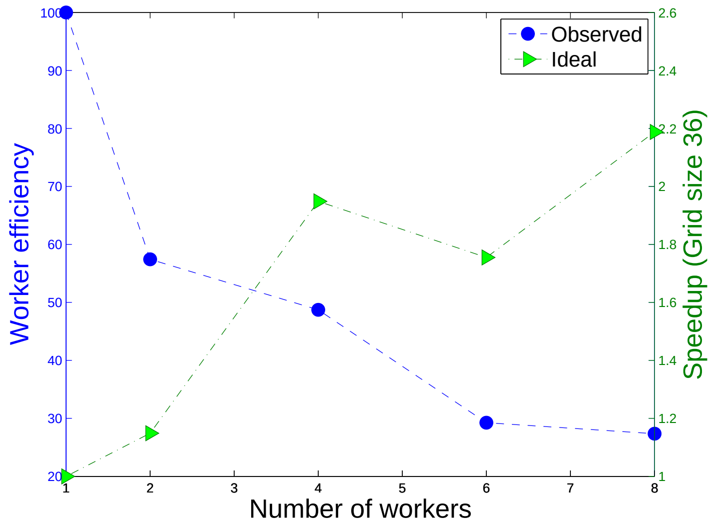


(c) Evolution of average diameter of a particle over time

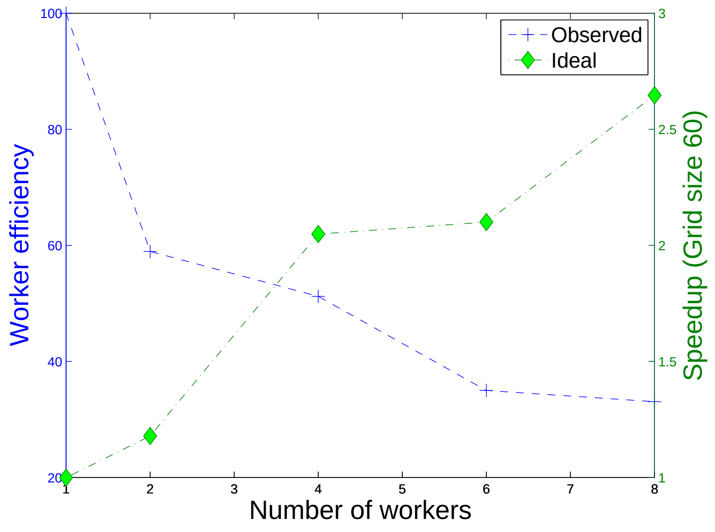
Figure 5.5 Comparison of temporal evolution of granule physical properties simulated for different worker pool classes,grid size=36



(a) Comparison of simulation times with increase in grid size for different worker pool classes



(b) Speedup and efficiency obtained for a grid size of 36



(c) Speedup and efficiency obtained for a grid size of 60

Figure 5.6 Plots of simulation times and obtained speedup of the PB code incorporating the `SPMD` construct

5.3 Speeding up a PBM code integrating more mechanisms

Profiler analysis

For the third case, we considered a more complex, integrated form of the PB code incorporating terms for consolidation, aggregation and liquid drying/rewetting. These mechanisms, in addition to breakage/attrition, are fundamental in describing the granulation process accurately to a greater extent. From Figure 5.7, it is evident that in spite of two additional mechanisms being present, aggregation remains the most computationally intensive, a characteristic that may be attributed to its 6 nested `for`-loops. Parallelization was achieved with the fork-join technique, a type of task parallelism. The `SPMD` keyword is used to force consecutive but independently executing sections of code to be split among the available pool of workers, followed by collection of calculated data at the end. The functions parallelized were those computing for drying/rewetting, consolidation, and finally aggregation (formation and depletion), each of which were assigned to run on individual workers, to improve parallelism.

Numerical accuracy validation

The temporal evolution of physical properties were plotted for both the SPMD and the single CPU versions of the code (Figure 5.8). As can be seen from Figure 5.8(a), the total number of particles predictably decrease over time due to aggregation by coalescence. The total volume of the particles, Figure 5.8b, on the other hand rises at a steady state as a result of continuous liquid binder addition over time and is also the reason why the average granule diameter increases gradually in Figure 5.8c. The tendency for these curves to level off after a certain period of time is due to the limited number of bins in the grid, 15, which restricts the the extent of granule aggregation and growth. This further serves to stress the need for faster simulations through parallelization in order to circumvent these restrictions and run the code for longer and for higher number of bins. Data for both the SPMD and single CPU versions are in good agreement with each other, affirming numerical precision and validity of the SPMD version results.

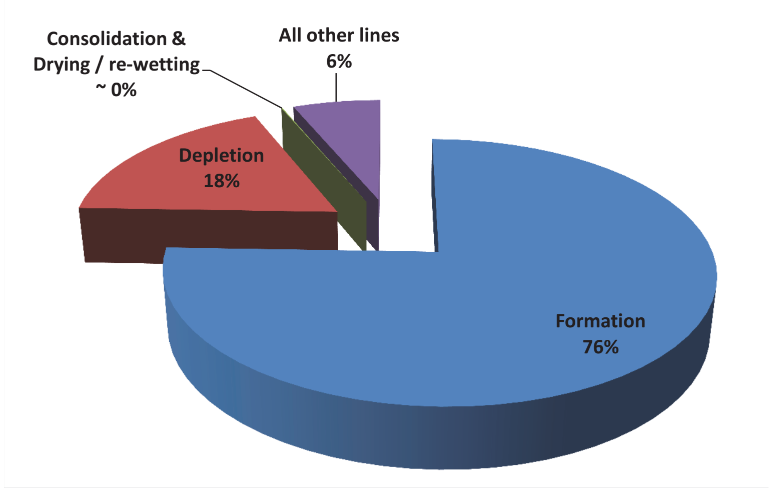
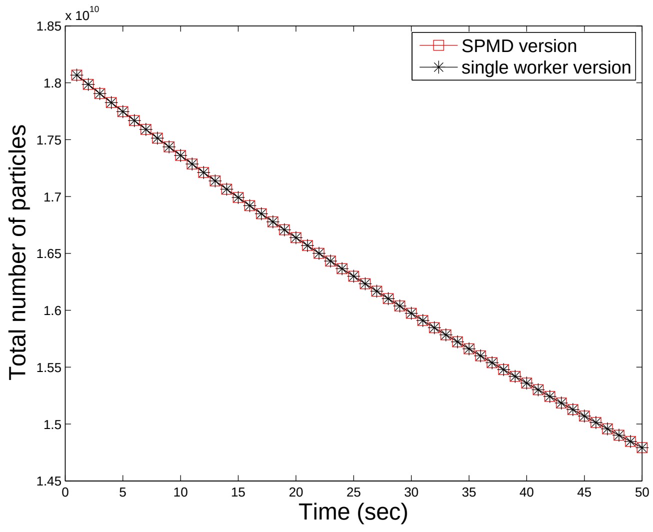


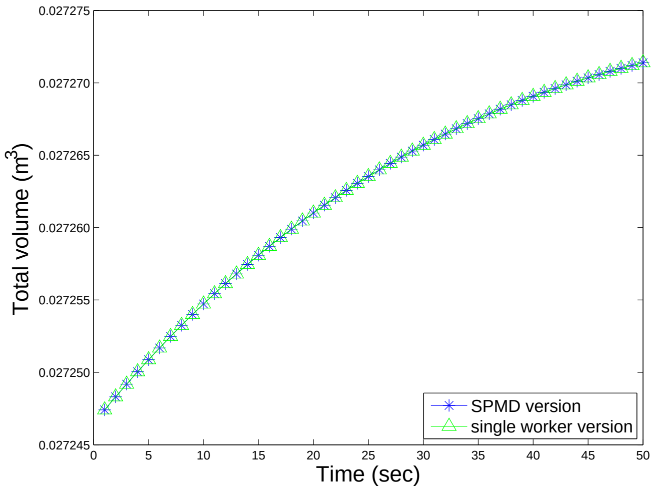
Figure 5.7 Piechart representation of MATLAB's profiler results for the 3-D granulation code with aggregation, consolidation, and liquid drying/rewetting run on a single lab.

Performance evaluation

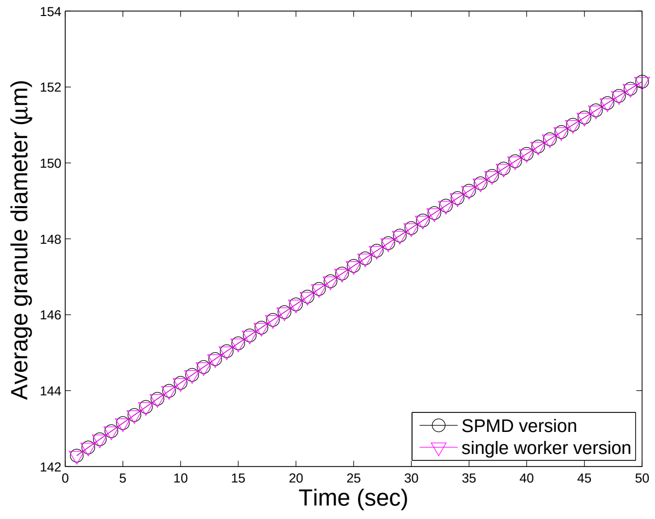
The simulation was carried out on the INTEL Core 2 Quad Q6600, utilizing all four cores. The grid size was varied and the corresponding simulation times plotted(Figure 5.9(a)). Even for a grid size of just 15 a speedup of 15.5 times was achieved, which is surprising considering how only four workers are used. The semi-log plot in Figure 5.9(b) is also shown to highlight the positive change in speedup beyond a grid size of 6. This is an example of *superlinear speedup*, where for n processors, a speedup of greater than n is produced.[87] Superlinear speedup is a special case, and may occur if problem size per processor is small enough to fit into registers, data caches or other smaller, yet faster memory banks instead of the on-board RAM.[88] Since some of the paralellized functions like drying/rewetting and consolidation utilize just a few variables per processor, causes of parallel inefficiency (load imbalance, interprocessor communication) are masked, resulting in faster multiplication-addition (MAD) operations than on a uni-processor machine, where bandwidth consumption would be higher than the rate at which RAM could deliver. This particular case proves that task-parallelism can indeed be useful in cases where the problem can be partitioned into sections capable of being executed independently and asynchronously in am embarrassingly-parallel manner.



(a) Evolution of total number distribution of particles over time

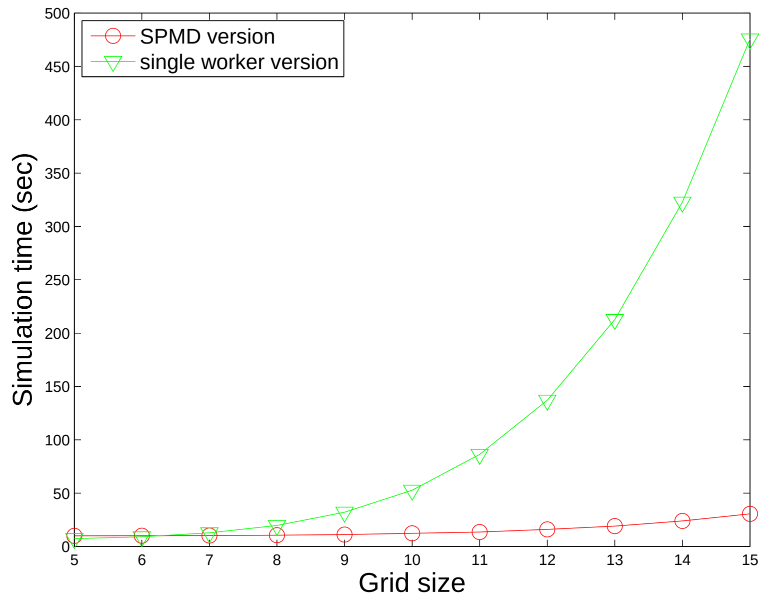


(b) Evolution of total volume of particles over time

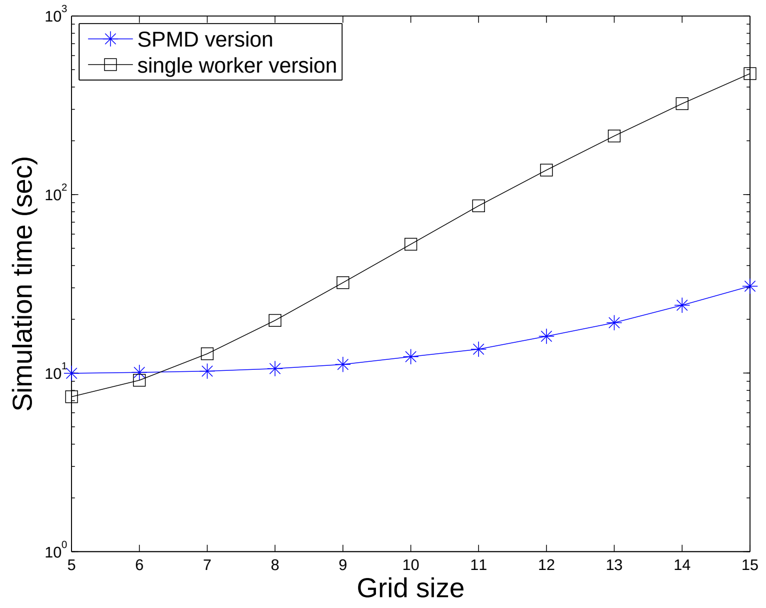


(c) Evolution of average diameter of a particle over time

Figure 5.8 Comparison of temporal evolution of granule physical properties for a sequential and parallel PBM code including consolidation and drying/rewetting, Grid size=15



(a) Plot comparing variation in simulation times of SPMD and single lab version with increasing grid size



(b) Semi-log plot comparing simulation times of SPMD and single worker version with increasing grid size highlighting positive speedup after a grid size of 6

Figure 5.9 Comparison of simulation times for a sequential and parallel PBM code

5.4 Distributed execution of a 3-D PBM code with breakage and cell averaging

Profiler analysis

The serial version of the 3D population balance code was developed as described in the previous section. After issuing the `profile on` command to initiate the MATLAB profiling utility, the script was executed on one worker. The profiler results are displayed in figure 5.10. As can be observed from the figure, the primary computationally intensive parts are those forming the core aggregation kernel: Solid, liquid and gas phase fraction relocating into adjoining bins (using the cell-average method), followed by formation and then depletion. Breakage and its associated functions took relatively much lesser time to compute, mainly due to the lack of integral terms. Previously, it was shown that formation was the primary bottleneck in a 3D PBM simulation using a linear grid. However, a non-linear grid would require much fewer bins than the linear grid to cover the same granule size range. This justifies the incorporation of the cell average method for relocating particle phase fractions to appropriate adjacent bins, albeit, at a slightly higher initial cost of computation. Once computational bottlenecks were identified,

the next step was to incorporate parallelism into the code. The data parallel approach was deemed ideal for this algorithm owing to the presence of nested `for` loops that perform numerous MAD (Multiply-ADd) operations on all elements of the same data set. Moreover, this divide-and-conquer strategy would likely scale well on distributed systems with shared memory. The loop slicing technique described in the previous section was utilized to implement data parallelism, effectively assigning the same task to operate on different partitions of the shared array concurrently. The aggregation and breakage functions were consolidated into a single function and the outermost `for` loop was sliced in accordance with the number of workers available (refer Equation 4.13), followed by a performance assessment for five cases: one(sequential), two, four, eight local workers and 8 distributed workers. For the 8 distributed worker case the parallel code was executed in a distributed manner using 4 cores per node. The distributed

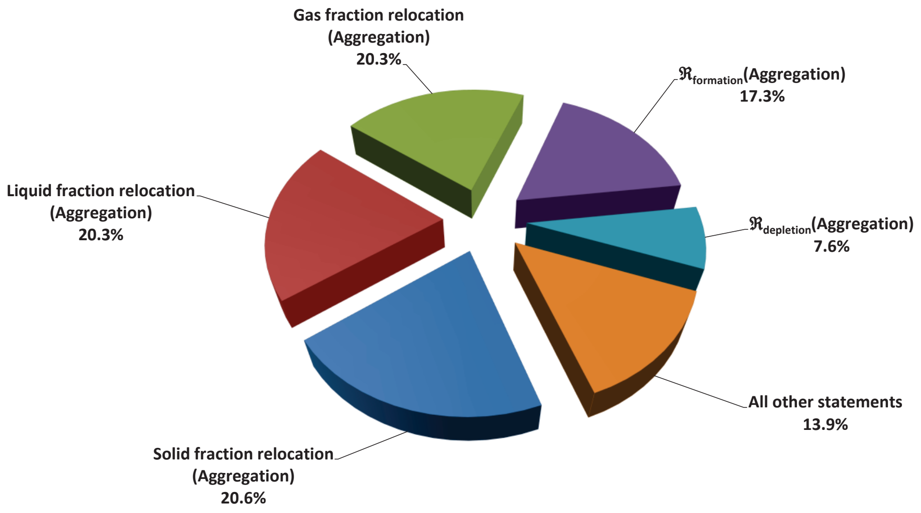
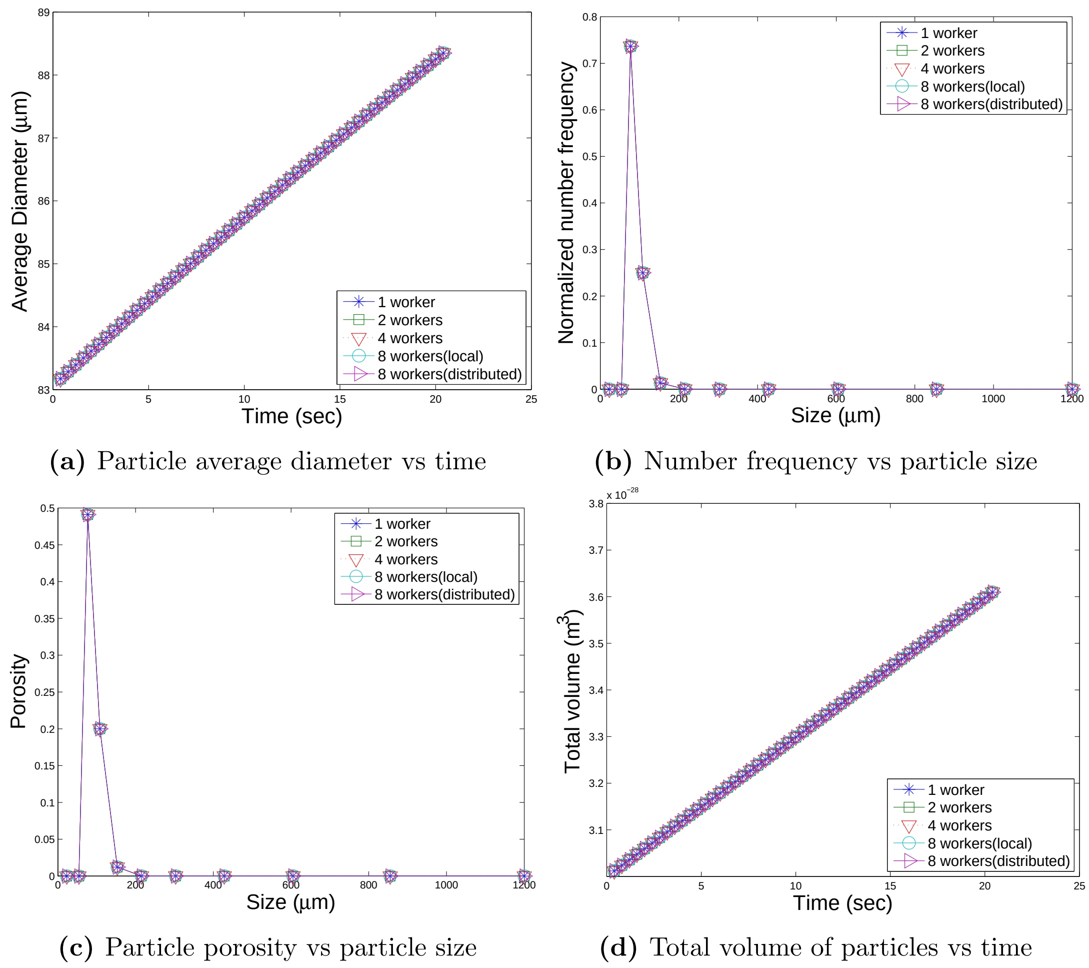


Figure 5.10 Piechart representation of MATLAB's profiler results for a serial version of the 3D granulation population balance code run on a single worker.

system consisted of two nodes linked via a high speed gigabit ethernet cable, each node housing a quad-core Core i7-2600 CPU running at 3.4GHz (stock), and 16GB of local memory (4 DIMMs $\times$ 4GB @ 1600 MHz).

Numerical accuracy validation

Speedup benefits of parallelization are meaningless if the simulation results are not reproducible and reasonably accurate, numerically. To verify this, the resulting data from each of the four parallel cases are superimposed to confirm numerical precision of the each parallel simulation case (see Figures 5.11(a),5.11(b),5.11(c),and 5.11(d)). Initial input parameters for the simulation are given in table 5.5. Figure 5.11(a) shows that the average diameter linearly rises due to the combined effects of liquid binder addition and aggregation, offsetting consolidation and breakage. Both particle number and porosity distribution (Figures 5.11(b) and 5.11(c)) is limited to the 0 - 100μm size class by the end of the short simulation period (t = 20 sec), while the increase in total volume (Figure 5.11(d)) at each time step is very minimal due to gradual liquid binder addition into the system. It is evident that the results from each parallel version conforms numerically to the results of the sequential version (1 worker) showing that computational accuracy was not compromised during distributed execution.



(a) Particle average diameter vs time

(b) Number frequency vs particle size

(c) Particle porosity vs particle size

(d) Total volume of particles vs time

Figure 5.11 Comparison of temporal evolution of particle physical properties for a sequential and parallel PBM code will cell average, Grid size=15

Performance evaluation

The most direct metric for measuring parallel performance is the speedup factor, and is generally represented as the ratio of serial execution time to parallel execution time (Equation 5.3). The speedup ratios thus calculated were plotted versus number of workers. In Figure 5.12, speedup is shown for a simulation that was run on a single CPU making use of 1, 2, 4, and 8 labs. These 8 labs were initiated on 8 threads present in the quad-core CPU (4 cores X 2 threads each) with the `matlabpool open` command. Another linearly rising plot in the same figure highlights the improvement in speedup after using 8 distributed workers running on 2 nodes over 8 labs on a single node. This comparison serves to show why assigning tasks to cores (equivalent to workers) provides better performance than assigning the same number of tasks to labs (equivalent to threads). This is not unexpected since memory transfer overheads for inter-thread communication are much higher than for cores.[55] A speedup of 3.79 was achieved with 8 labs on a single CPU (an improvement of 94.7%), which is very close to the theoretical maximum speedup possible i.e 4 times (corresponding to 4 cores). On a distributed system, this performance increase is pushed even higher with a peak speedup of 6.21 times on 8 workers (77.61% improvement). The ideal speedup line shown in both figures indicates the theoretical maximum for each worker pool set and is equal to the number of physical cores (workers) available. It must be kept in mind that there are almost always some statements in a parallel algorithm that cannot be parallelized, and have to be executed serially on one worker. This serial fraction (f) limits the maximum speedup attainable with a certain pool of workers (n), given by equation 5.5, also known as Amdahl's law.[89]

$$S(n) = \frac{n}{1 + (n-1)f} \tag{5.5}$$

The maximum speedup according to Amdahl is calculated for each parallel case and plotted in Figure 5.12. From the graph it is clear that the observed speedup factor line is initially in proximity with the maximum speedup line, but gradually diverges as the number of workers increase. This leads us to the issue of scalability, a property of cluster systems exhibiting linearly proportional increase in performance of the parallel algorithm with corresponding increase in system size (i.e. addition of more processors).[90] Several metrics have been proposed to synthetically quantify scalability, and due to the pre-determined nature of the initial problem size, the *fixed serial work per processor* approach based on *efficiency* was adopted, and the resulting speedup is termed *scaled speedup*. It relies on the problem size (number of bins in a grid) remaining constant even as the number of processors are increased. Only if the memory overheads associated with distributed execution rise linearly as a function of n, and parallel execution time (T_n) keeps on decreasing, will the algorithm be considered scalable. Efficiency (equation 5.4), in its fundamental form is defined as the fraction of time that workers actually take to perform computations, and is expanded in equation 5.6.

$$E_n = \frac{T_s}{T_n \times n} = \frac{S(n)}{n} \times 100 \tag{5.6}$$

where T_s is the serial execution time. A more descriptive definition of efficiency may be given as:[91]

$$E_n = \frac{t_c W}{n T_n} = \frac{t_c W}{t_c W + T_0(n, W)} \tag{5.7}$$

where E_n is efficiency of the system size n; $T_0(n, W)$ the total overhead of the distributed system; t_c the average execution time per operation in the architecture and is a constant; W is the problem size, which translates to processor work; and $t_c W$ is the serial computing time (T_s) of an algorithm. $T_0(n, W)$ is calculated as:

$$T_0 = pT_n - T_s \tag{5.8}$$

Since t_c is a constant and depends on the underlying architecture itself, it can be calculated from processor specifications. We used a Intel Core i7 'Sandybridge' CPU which maintained a 3.512 GHz clock during the simulations i.e, 3.512×10^9 cycles/second. In the Sandybridge architecture, 4 floating point operations (flops) can be computed per clock cycle, which allows us to calculate the maximum theoretical flops that can be processed in a second (v): 1 core $\times$ 4 flops/core/cycle $\times$ 3.512 $\times 10^9$ cycles/sec = 14.046 gigaflops/sec; therefore the time required to compute 1 flop, t_c is 7.119 $\times 10^{-11}$ seconds. Using this information we can determine the serial work W from equation 5.7. Parallel work W_n is defined as:

$$W_n = n \times T_n \times v \tag{5.9}$$

See table 5.2 for the calculated values of efficiency, parallel overhead and processor work. Extracting the expressions for T_s and T_n from equations 5.7 and 5.9 respectively

Table 5.2 *Table of performance evaluation metrics for the parallel 3-D PBM simulation with cell averaging*

Workers	T_0(sec)	E_n	W_n (flops)
1	–	1	2.149×10^{14} (serial work, W)
2	919.3	0.9433	2.278×10^{14}
4	3405.9	0.8179	2.627×10^{14}
8(local)	16992.1	0.47	4.536×10^{14}
8(distributed)	4409.3	0.776	2.768×10^{14}

and combining them with equation 5.6 yields: $E_n = \frac{W}{W_n}$. It can be inferred that E_n cannot exceed 1 and addition of workers beyond n does not increase S_n if parallel work W_n exceeds the serial work W. From table 5.2, it is apparent that parallel work is indeed consistently greater than serial work for increasing worker counts, supporting this observation. Even increasing the number of workers to infinity only results in bringing down efficiency closer to zero (refer equation 5.6). Therefore, for constant-problem sized scaling, the speedup does not continue to increase with the increasing number of processors, but tends to saturate or peak at a certain system size, a principle apparent from the trajectory of the speedup curve (Figure 5.12). Thus the 3-D granulation population balance algorithm cannot be perfectly scalable for a fixed-size problem. According to Amdahl, even with an infinite number of processors the maximum speedup for a code is restricted to $1/f$, the inverse of its serial fraction, which yields a value of 83.33 times for our simulation. This implies that utilizing more than 84 workers will not produce any additional improvement in performance. Even this limit is virtually impossible to reach, as Amdahl's equation does not take into account memory transfer overheads, effects due to improper load balancing and synchronization, all of which become more dominant as the number of workers increase. Moreover, to remain scalable, W_n must be a function of the number of processors, n, to ensure that parallel overhead T_0 does not grow faster than n rises.[91] This is where another metric ,the *overhead ratio*, proves useful for testing the performance of our parallel algorithm with additional workers. This concept is introduced in the next section, followed by an analysis of this ratio for both 3-D and 4-D granulation codes with a view to compare scalabilities.

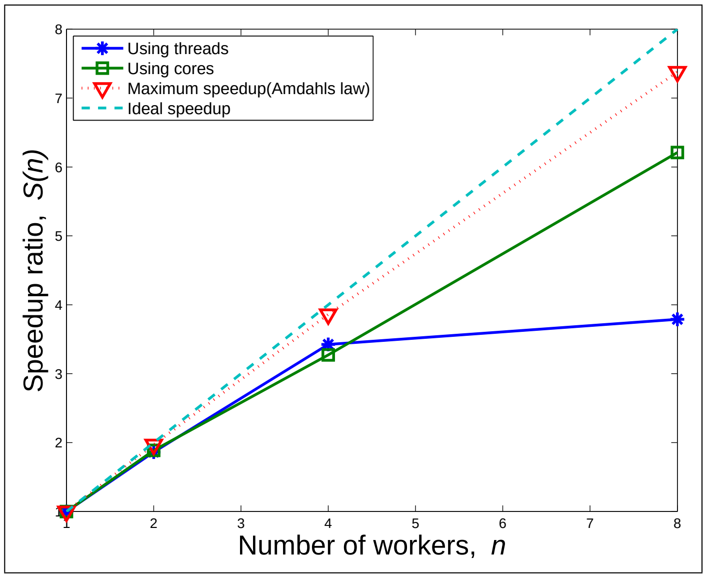


Figure 5.12 Comparision of speedup using a 8 threads on a single CPU, 8 cores on 2 nodes, and maximum speedup as predicted by Amdahl's law. The dashed line represents the theoretical upper bound for speedup and is equal to the number of available workers.

5.5 Distributed execution of 4-D PBM code with breakage and cell averaging

Profiler analysis

The serial version of the 4 dimensional population balance model code was built following the procedure described in chapter 4. The initial grid size was set to 8 along each dimension, with a process run time of 20 seconds in order to keep actual execution time within a feasible range sufficient for comparison. After debugging, some sections were vectorized to partially reduce the overall execution time and concentrate burdensome computations on aggregation and breakage. Using the MATLAB profiler utility, a breakdown of the time spent calculating each statement was obtained and the results charted (Figure 5.13). Once again, it is clear that aggregation and its associated calls are the main bottlenecks in the code: Solid, liquid and gas phase fraction relocation into adjoining bins using cell-average take the most time, followed by formation due to aggregation. Aggregation-induced depletion consumes nearly 10% of the total run time, while breakage and its associated calls took relatively much lesser time to compute. The second solid component ('solid 2') present in the initial distribution adds further computational complexity to our 4D PBM code as it introduces another pair of `for`-loops to cover the entire grid range. In all, there are eight nested `for`-loops accounting for the integral terms (Equation 2.14) making it the candidate of choice for parallelism via loop slicing. The outer loop enclosing the aggregation and breakage functions was sliced in accordance with Equation 4.13 to ensure data-parallel execution on 8 workers.

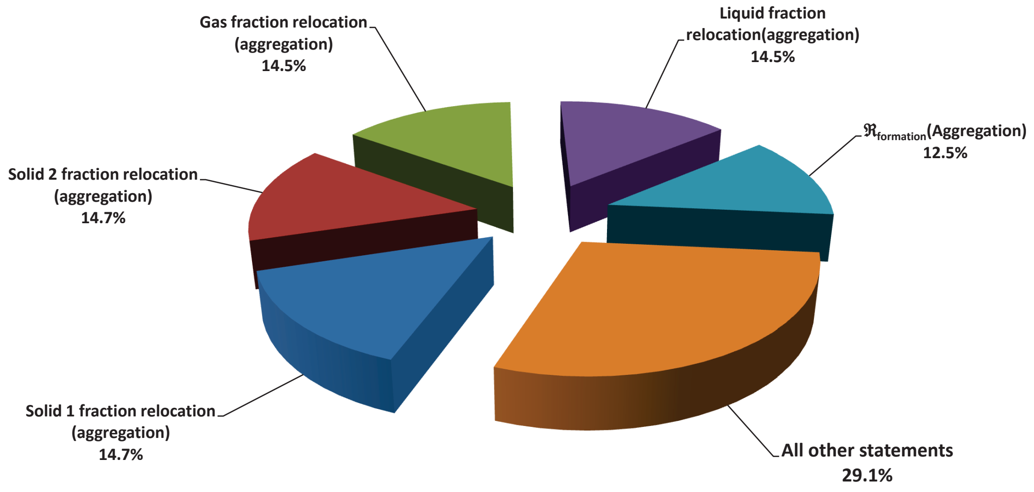
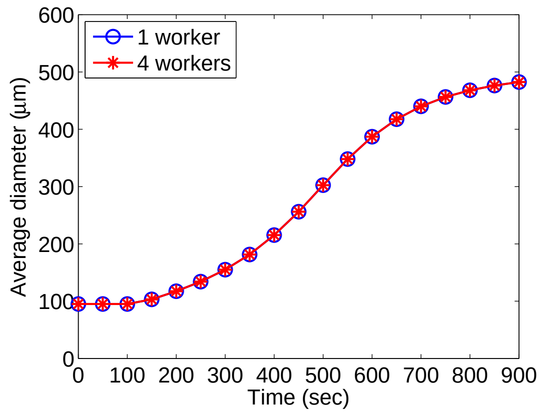


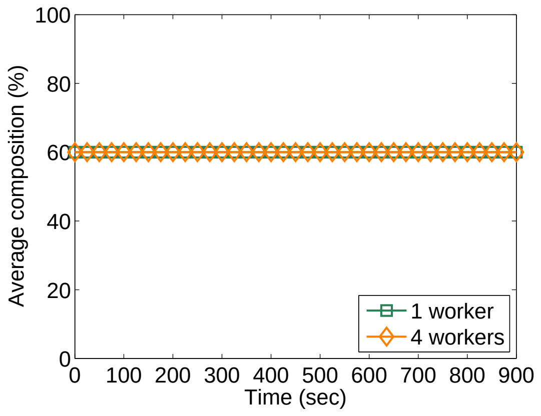
Figure 5.13 Piechart representation of MATLAB's profiler results for a serial version of the 4D granulation population balance code run on a single worker.

Numerical accuracy validation

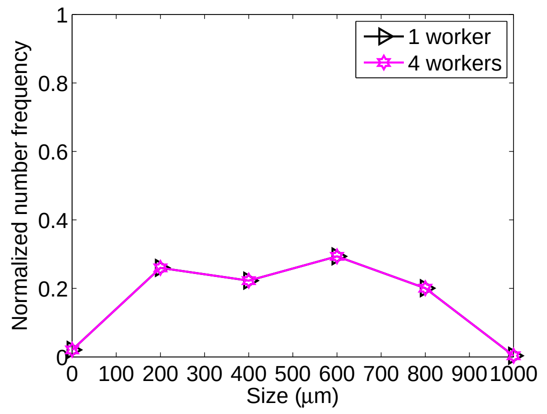
To confirm numerical precision of the parallel simulation results for a 4-D PB code, a more accurate set of initial parameters was used (see Table 5.5) instead of those for profiling and performance analysis. This was done to validate the effectiveness of our parallel model with approximately realistic parameters. In addition, the results were compared for only 2 cases - 1 and 4 workers - due to our trial version of the MDCS toolbox expiring soon after. Results from each case were directly superimposed as in the case of the 3-D model to check for consistency of numerical precision (see Figures 5.14(a),5.14(b),5.14(c), and 5.14(d)). By visual inspection it is evident that the data for



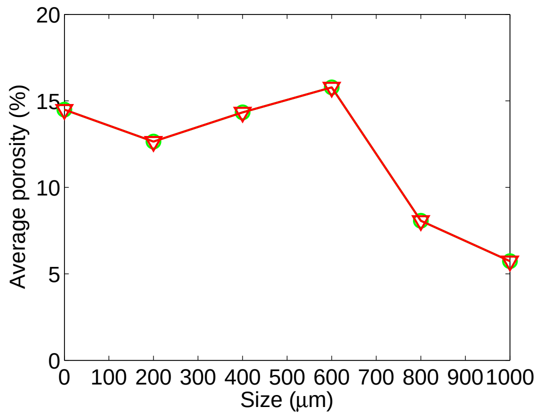
(a) Evolution of average diameter of a particle over time



(b) Average composition distribution of solid component 1 (s_1)



(c) Evolution of total volume of particles over time



(d) Average porosity distribution of a particle

Figure 5.14 Comparison of temporal evolution of particle physical properties for a 4-D sequential and parallel PBM code will cell average, Grid size=15

granule properties in the 4 worker parallel case is entirely identical to the data from the serial version (1 worker), confirming that computational accuracy was not compromised during distributed execution. Figure 5.14(a) shows the evolution of particle diameter over time. It linearly rises due to the combined effects of liquid binder addition and aggregation, offsetting consolidation and breakage phenomena. This brought up the average diameter from under 100 μm to around 500 μm. Distribution of the average composition of solid component 1 (s_1) remains constant (Figure 5.14(b)) because of the conservation of mass with respect to s_1 in the system. Particle size distribution after granulation, weighted by volume and normalized, is presented in Figure 5.14(c). Average particle porosity is directly affected by the volumes of liquid and gas in a particle, and the distribution is plotted in Figure 5.14(d).

Performance evaluation

The speedup ratios were calculated from Equation 5.3 and plotted against the number of workers. As shown for the 3D case, running on a distributed system produced better performance as opposed to execution on threads, since distributed workers can create local arrays simultaneously, saving transfer time.[53] Hence, there was no need to perform a speedup comparison between cores and threads for this particular case. Eight workers were initiated on 8 cores divided across two nodes (4 cores per node) with the `matlabpool open` command. Each core on the INTEL Core i7 2600 CPU has a theoretical SSE (Streaming SIMD Extensions) rate of approximately 14 gigaflops per sec as computed in the previous section. Figure 5.15 shows the speedup for a simulation run on the distributed system making use of 1, 2, 4, and 8 workers. Initial input parameters for the simulation are given in table5.4. The linearly rising plot shows a peak speedup of 5.6 times over a single worker, using 8 workers. The maximum permissible speedup as calculated according to Amdahl's law is roughly 7.6 times considering a non-parallelizable, serial fraction of 0.8% of the total execution time. The ideal speedup line shown in both figures indicates the theoretical maximum for each worker pool set and is equal to the number of physical cores (workers) available. The observed speedup line diverges from the theoretical ideal and Amdahls's speedup much more

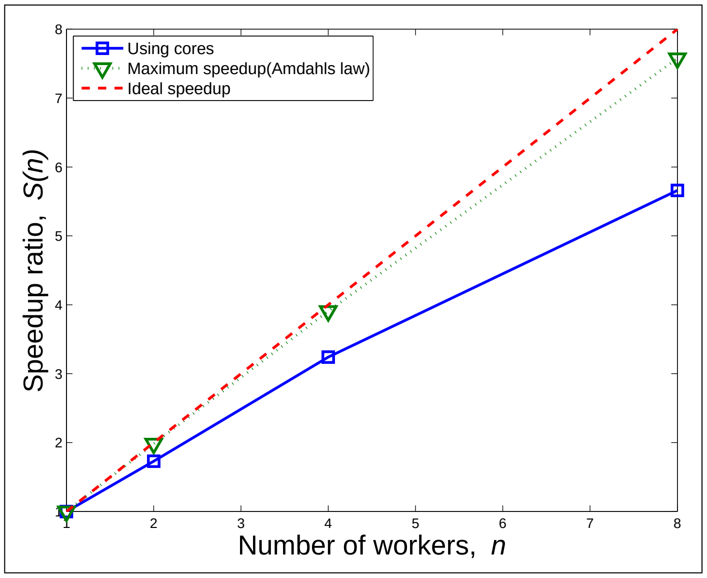


Figure 5.15 Comparision of speedup with 8 workers on 2 nodes, and maximum speedup as predicted by Amdahl's law. The dashed line represents the theoretical upper bound for speedup and is equal to the number of available workers.

rapidly than for the 3-D granulation simulation case, a trend evident in the speedup plot (Figure 5.15). As a result, we expect the scalability of the algorithm to naturally decrease as more nodes housing CPUs of the same specifications are added. To quantify scalability, we once again rely on the fixed-problem size scaled approach since the initial 4D grid size will remain the same even when the available pool of workers are increased. The algorithm is considered scalable only if the overheads associated with data transfer between the job manager and workers rise only linearly as a function of the number of workers n.[92] While discussing scalability, an aspect to be considered is the scalability of an algorithm with respect to underlying hardware. Algorithms scalable on one architecture need not necessarily scale well on others, and hence deploying the application on heterogeneous cluster systems may prove to be counter-productive. Since our distributed system is homogeneous in terms of underlying hardware, we can safely assume that if the algorithm is scalable on the existing system, it will also scale well on additional workers of the exact same specifications. An important measure of such an algorithm's efficiency is its *overhead ratio* - a ratio of its communication overhead to parallel execution time (Equation 5.10). The lower the ratio, the more each worker will perform effectively. Typically, this ratio increases swiftly with increasing number of workers but decreases as the problem size grows.[90] For the specific case of the 4-D granulation simulation, this intrinsic property of a parallel algorithm presents the best means of explaining the accelerated decline in speedup as worker counts increase.

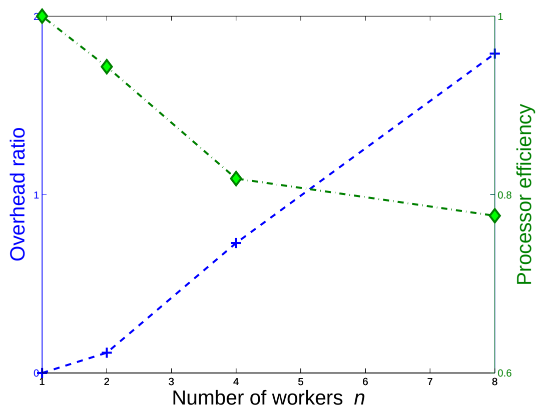
$$\text{Overhead ratio, } \varnothing = \frac{\text{Parallel overhead}}{\text{Parallel execution time}} = \frac{T_0}{T_n} \tag{5.10}$$

Table 5.3 displays the calculated values for efficiency, parallel overhead in seconds and processor work in flops. From the relation $E_n = \frac{W}{W_n}$, it can be inferred that E_n cannot exceed 1 and addition of workers beyond n will not increase speedup if parallel work, W_n, remains greater than the serial work, W. From the table it is evident that parallel work is indeed consistently greater than serial work for increasing worker counts, proving that perfect scalability is unattainable for the 4-D PBM. Moreover, W_n must remain a function of the number of processors, n, to ensure that parallel overhead T_0 does not grow faster than n rises.[91] To observe this trend, we plot the overhead ratio against the number of workers, as seen in Figure 5.16(b). A relatively linear rise of parallel overhead with the number of processors is observed for 8 workers accompanied by the inevitable decrease in processor efficiency. On comparison with the 3-D granulation simulation (Figure 5.16(a)), we find the overhead ratio is approximately 3 times lesser than for the 4-D code across 8 workers. The most apparent reason for this difference is that the time required for distribution of 4 dimensional arrays across workers is much higher than for 3 dimensional data arrays. Larger, multi-dimensional arrays have a higher memory requirement during data creation and modification, entailing longer read/write times over the network. Thus, we can classify this 4-D PBM simulation

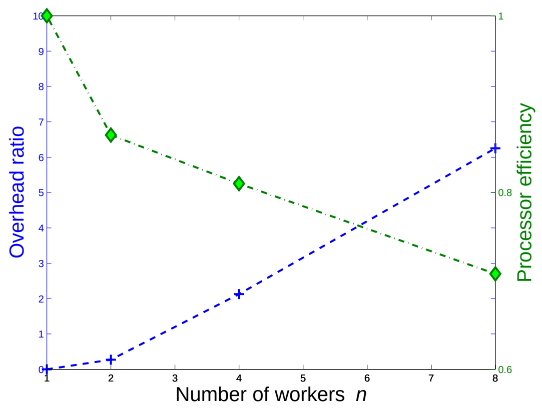
Table 5.3 *Table of performance evaluation metrics for the parallel 4-D population balance model simulation*

Workers	T_0(sec)	E_n	W_n (flops)
1	–	1	7.24×10^{13} (serial work, W)
2	749.9	0.865	8.36×10^{13}
4	1207.1	0.81	8.94×10^{13}
8	2131.05	0.708	1.02×10^{14}

as a case of memory-constrained parallelism. This additional access overhead alone consumes a significant portion of useful processor time especially as the input grid size increases, bringing down processor efficiency drastically. This is apparent from the nearly 10% drop in efficiency from using a 3-D to 4-D grid for 8 workers (see Figures 5.16(a) and 5.16(b)). Following Amdahl's hypothesis, an infinite number of processors will only yield a maximum speedup of $1/f$, the serial fraction, which yields a value of 125 times for the 4-D case, implying that utilizing more than 125 workers will not produce any additional improvement in performance. Nevertheless, this value is a gross overestimate as explained in the previous section, and more so for the 4-D case as Amdahl's equation does not take into account memory transfer overheads, the performance-deteriorating effects of which have just been demonstrated.



(a) Variation of overhead ratio with processor efficiency as the number of workers are increased for the 3-D population balance simulation



(b) Variation of overhead ratio with processor efficiency as the number of workers are increased for the 4-D population balance simulation

Figure 5.16 Comparison of overhead curves for 3-D and 4-D distributed PBM code with cell averaging

Table 5.4 *Process parameters and initial conditions used in 3-D PBM simulation.*

Parameter name	**Value**
ρ_{solid}	$2700\ kg/m^3$
ρ_{liquid}	$1000\ kg/m^3$
ρ_{gas}	$1.2\ kg/m^3$
Granulation time	$20\ s$
Time step size	$0.4\ s$
Volume of first bin, solid component $s_{1,1}$	$1 \times 10^{-13}\ m^3$
Volume of first bin, liquid binder l_1	$2 \times 10^{-13}\ m^3$
Volume of first bin, gas g_1	$1 \times 10^{-13}\ m^3$
Total number of bins of in each dimension	16
Initial particle count F in bin $(1,1,1)$	$1 \times 10^{-15}\ mol$
Aggregation constant β_0	$8 \times 10^{14}\ mol^{-1}s^{-1}$
Aggregation constant α	1
Aggregation constant δ	1
Breakage constant P_1	$7 \times 10^{-11}\ m^{-1}$
Breakage constant P_2	$1.3\ m^{-1}$
Shear rate G_{shear}	$60\ s^{-1}$
Consolidation rate constant c	$1 \times 10^{-21}\ s^{-1}$
Minimum porosity ϵ_{min}	0.2
c_{binder}	0.1
Liquid binder spray rate $\dot{V}_{spray}$	$5 \times 10^{-3}\ m^3/s$

Table 5.5 *Process parameters and initial conditions used in 4-D PBM simulation.*

Parameter name	**Value**
Granulation time	$900\ s$
Time step size	$0.5\ s$
Volume of first bin, solid component 1 $s_{1,1}$	$1 \times 10^{-13}\ m^3$
Volume of first bin, solid component 2 $s_{2,1}$	$1 \times 10^{-13}\ m^3$
Volume of first bin, liquid binder l_1	$2 \times 10^{-14}\ m^3$
Volume of first bin, gas g_1	$1 \times 10^{-14}\ m^3$
Total number of bins of in each dimension	8
Initial particle count F in bin $(1, 2, 1, 2)$	$3 \times 10^{-13}\ mol$
Initial particle count F in bin $(2, 1, 1, 2)$	$7 \times 10^{-13}\ mol$
Aggregation constant β_0	$1 \times 10^{19}\ mol^{-1}s^{-1}$
Aggregation constant α	1
Aggregation constant δ	1
Breakage constant P_1	$1\ m^{-1}$
Breakage constant P_2	1
Shear rate G_{shear}	$85\ s^{-1}$
Consolidation rate constant c	$1 \times 10^{-7}\ s^{-1}$
Minimum porosity ϵ_{min}	0.1
Liquid binder spray interval	$120\ s < t < 360\ s$
Liquid binder spray rate $\dot{V}_{spray}$	$25\ mL/s$

Chapter 6

Conclusions and recommendations for future work

Parallel computing has been a subject of intense study for several years, but its application to particulate processes described by population balance models has been limited to a few studies in crystallization[26].[25] This study has explained the development and implementation of parallel algorithms describing granulation behaviour on 3 hardware platforms: a multi-core CPU; a many-core GPU; and a dual-node distributed computing system. The procedure followed herein for parallelizing PBMs involved three steps: locating portions of the code that are most time-consuming with tools like MATLAB profiler; applying one of the two paradigms for parallelism as appropriate; and finally, optimizing for minimal variable transfer overhead. We have proposed here, two methods of efficiently parallelizing the integro-differential PBEs that make up the aggregation term for a multi-core CPU, either using loop-slicing, or alternately, via a brute force, fork-join method in conjunction with MATLAB's Parallel Computing Toolbox. This approach to parallelism provides the easiest approach to reducing simulation times. The results show a speedup of 2.6 times with 8 labs over a sequential code.

For the first time, a method describing the utility of GPU computing for PBMs is demonstrated. The JACKET toolbox for MATLAB was utilized to efficiently parallelize `for`-loops across the 240 cores on a single NVIDIA GTX 280 card. Although the performance advantage of the GPU over CPU did not seem encouraging initially, due to lower clock frequency and on-board memory and JACKET's own restrictions, a closer analysis of speedup ratios revealed that the GPU has potential to outclass the CPU at very large grid sizes, especially given the significant advances made in its architecture with each new generation. Thirdly, a relatively more complex code, integrating several mechanisms, was parallelized with the `SPMD` keyword, yielding a superlinear speedup of

15.5 times.

For distributed computing, two processes were modeled based on population balance principles: a single-component granulation process represented by a three-dimensional PBM; and a multi-component granulation process represented by a four-dimensional PBM. Both models incorporated a breakage mechanism and a cell averaging technique to re-allocate daughter particles in adjacent bins. Profiling both codes showed that the cell average technique consumed the most time and needed to be parallelized. Following a distributed shared memory approach to achieve data parallelism, the models were parallelized with the loop slicing technique. The MATLAB toolboxes utilized for this purpose were the Parallel Computing Toolbox in conjunction with the Distributed Computing Server. The results show a good speedup of 6.2 times with 8 workers for the 3-D PBM code while the 4-D code was slightly less efficient with a speedup factor of 5.7 times.

It was shown that perfect scalability is theoretically impossible to attain for either PBM, since fixed-problem size scaling does not allow for 100% per processor efficiency, which in turn is due to serial work being consistently overwhelmed by parallel work. By Amdahl's law, even with an infinite number of processors, the speedup is limited by the fraction of algorithm that has not been parallelized. With the aid of the overhead ratio we compared scalabilities of the two algorithms on our homogeneous distributed system. While the overhead ratio rose linearly with respect to the number of workers in both cases, the performance is clearly better for the 3-D model which also displayed a higher processor efficiency with 8 workers. This is is usually a good indication that the 3-D algorithm will scale better than the 4-D, with a lower execution time. The 3-D code is predicted to yield no speedup benefit beyond 83 workers, whereas the speedup for 4-D code will be restricted to less than 125 workers. Amdahl's equation does not take into account memory transfer overheads, improper load balancing and synchronization, the effects of which becomes more dominant as the number of workers increase. In all parallel cases, simulation results were shown to be equal to the sequential versions' confirming that numerical precision was not compromised during distributed execution.

Future work will include building better parallel algorithms with efficient task scheduling for greater speedup, utilizing next-generation Tesla architecture-based NVIDIA GPUs. To further reduce transfer overheads in both the parallel and distributed algorithms, memory pre-allocation may be done using distributed data types and employing MPI-based constructs like `labSend` and `labReceive` for better memory read/write patterns. This will also offset the higher demand for system RAM that comes with utilizing larger arrays and/or increasing the dimensionality of the system.

GPU computing is still in its infancy with regard to MATLAB, primarily due to limited function support. With advances in many-core architectures - NVIDIA's Kepler, GPUs integrated into CPUs (AMD Fusion, INTEL HD 3000), newer devices like INTEL's Xeon Phi coprocessor - and their support for MATLAB increases, simulation times can be further reduced. This is due to the fact that once parallelized, PB models are well suited for execution on massively parallel architectures. Methods developed for CPU and GPU parallel computing can also be easily extended to other particulate processes described by PBMs such as crystallization, milling, blending and polymerization, with potential to aid computer-aided modeling and simulation, offering invaluable economic benefit to industries that deal with such processes.

Bibliography

1. Chen, C.-C. and Mathias, P. M. *AIChE Journal* **48**(2), 194–200 (2002).

2. Schmeisser, M., Heisen, B. C., Luettich, M., Busche, B., Hauer, F., Koske, T., Knauber, K.-H., and Stark, H. *Acta Crystallographica Section D* **65**(7), 659–671 Jul (2009).

3. Wilkinson, B. and Allen, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 1 edition, (1999).

4. Cundall, P. A. and Strack, O. D. L. *Geotechnique* **29**(1), 47–65 (1979).

5. Matthews, H., Miller, S. M., and Rawlings, J. B. *Powder Technology* **88**(3), 227–235 (1996).

6. Ramachandran, R., Immanuel, C. D., Stepanek, F., Litster, J. D., and Doyle III, F. J. *Chemical Engineering Research and Design* **87**(4), 598 – 614 (2009).

7. Muzzio, F. J., Shinbrot, T., and Glasser, B. J. *Powder Technology* **124**(1-2), 1–7 (2002).

8. Iveson, S. M., Litster, J. D., H., K., and Ennis, B. J. *Powder Technology* **117**(1), 3 – 39 (2001).

9. Gantt, J. A., Cameron, I. T., Litster, J. D., and Gatzke, E. P. *Powder Technology* **170**(2), 53 – 63 (2006).

10. Immanuel, C. D. and Doyle III, F. J. *Powder Technology* **156**(2-3), 213 – 225 (2005).

11. Poon, J. M.-H., Immanuel, C. D., Doyle III, F. J., and Litster, J. D. *Chemical Engineering Science* **63**(5), 1315–1329 (2008).

12. Dosta, M., Heinrich, S., and Werther, J. *Powder Technology* **204**, 71–82 (2010).

13. Ramachandran, R. and Barton, P. I. *Chemical Engineering Science* **65**(16), 4884 – 4893 (2010).

14. Ramachandran, R. and Chaudhury, A. *Chemical Engineering Research & Design* **Accepted**, doi:10.1016/j.cherd.2011.10.022 (2011).

15. Ramachandran, R., Ansari, M. A., Chaudhury, A., Kapadia, A., Prakash, A. V., and Stepanek, F. *Chemical Engineering Science* **Accepted**, doi: 10.1016/j.ces.2011.11.045 (2011).

16. Gantt, J. A. and Gatzke, E. P. *AIChE Journal* **52**(9), 3067–3077 (2006).

17. Stepanek, F., Rajniak, P., Mancinelli, C., Chern, R., and Ramachandran, R. *Powder Technology* **189**(2), 376 – 384 (2009).

18. Rajniak, P., Stepanek, F., Dhanasekharan, K., Fan, R., Mancinelli, C., and Chern, R. T. *Powder Technology* **189**, 190–231 (2009).

19. Freireich, B., Li, J., Litster, J., and Wassgren, C. *Chemical Engineering Science* **66**(16), 3592 – 3604 (2011).

20. Ramachandran, R., Arjunan, J., Chaudhury, A., and Ierapetritou, M. *Journal of Pharmaceutical Innovation* **6**, 249–263 (2011).

21. Raeth, P. and Chaves, J. In *High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2010 DoD*, 438–441, June (2010).

22. Panuganti, R. *A High Productivity Framework for Parallel Data Intensive Computing in MATLAB*. PhD thesis, The Ohio State University, (2009).

23. Swinburne, R. January (2011).

24. Zhang, Y., Mueller, F., Cui, X., and Potok, T. *Journal of Parallel and Distributed Computing* **71**(2), 211–224 (2011).

25. Gunawan, R., Fusman, I., and Braatz, R. D. *AIChE Journal* **54**(6), 1449–1458 (2008).

26. Ganesan, S. and Tobiska, L. *Chemical Engineering Science* (2011).

27. Salman, A. D., Seville, J. P., and Hounslow, M. *Granulation*, volume 11 of *Handbook of Powder Technology Series.* Elsevier Science, (2007).

28. Kapur, P. C. and Fuerstenau, D. W. *Industrial & Engineering Chemistry Process Design and Development* **8**(1), 56–62 (1969).

29. Annapragada, A. and Neilly, J. *Powder Technology* **89**, 83 – 84 October (1996).

30. Iveson, S. M. *Powder Technology, Control of Particulate Processess IV* **124**(3), 219 – 229 (2002).

31. Verkoeijen, D., Pouw, G. A., Meesters, G. M. H., and Scarlett, B. *Chemical Engineering Science* **57**(12), 2287–2303 (2002).

32. Ramkrishna, D. *Population balances: Theory an applications to particulate systems engineering.* Elsevier Science, (2000).

33. Adetayo, A. A. and Ennis, B. J. *AIChE Journal* **43**(4), 927–934 (1997).

34. Hounslow, M. *KONA* **16**, 179–193 (1998).

35. Sastry, K. V. *International Journal of Mineral Processing* **2**(2), 187 – 203 (1975).

36. Madec, L., Falk, L., and Plasari, E. *Powder Technology* **130**(13), 147 – 153 (2003).

37. Pinto, M. A., Immanuel, C. D., and Doyle, F. J. *Computers and Chemical Engineering* **31**(10), 1242 – 1256 (2007).

38. Bilgili, E. and Scarlett, B. *Powder Technology* **153**(1), 59 – 71 (2005).

39. Pandya, J. and Spielman, L. *Chemical Engineering Science* **38**(12), 1983 – 1992 (1983).

40. Lee, K., Kim, T., and Rajniak, P. *Chemical Engineering Science* **63**, 1293–1303 (2008).

41. Marshall, C. L., Rajniak, P., and Matsoukas, T. *Chemical Engineering Research and Design* **In Press, Corrected Proof**, – (2010).

42. Jr., C. L. M., Rajniak, P., and Matsoukas, T. *Powder Technology* (0), – (2012).

43. Moore, G. E. *Electron. Mag.* **38**(8), 114–117 (1965).

44. University of Notre Dame (2012, M. . Online, May (2012).

45. ORACLE. Technical report, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, May (2010).

46. SPEC. *Third Quarter 2011 SPEC CPU2006 Results.* Standard Performance Evaluation Corporation, 7001 Heritage Village Plaza, Suite 225, Gainesville, VA 20155, September (2011).

47. Grama, A., Gupta, A., Karypis, G., and Kumar, V. *Introduction to Parallel Computing.* Addison Wesley, 2 edition, January (2003).

48. Foster, I. *Designing and building parallel programs.* Addison Wesley, 1 edition, January (1995).

49. Barney, B. August (2011).

50. Gao, W. and Kemao, Q. *Optics and Lasers in Engineering* (2011).

51. Duncan, R. *Computer* **23**, 5 – 16 February (1990).

52. Siewert, S. Technical report, Atrato, Inc., december (2009).

53. MathWorks. *Parallel Computing Toolbox: Product description.* MathWorks Inc., 3 Apple Hill Drive, Natick, MA 01760-2098, USA, September (2011). Online.

54. Luszczek, P. *International Journal of High Performance Computing Applications* **23**, 277–283 (2009).

55. Warg, F. *Techniques to Reduce Thread-Level Speculation Overhead.* PhD thesis, Department of Computer Science and Engineering, Chalmers University Of Technology, (2006).

56. Gropp, W., Lusk, E., and Skjellum, A. *Using MPI : Portable programming with the Message-Passing Interface.* Massachusetts Institute of Technology press, 2 edition, (1999).

57. Peleg, D. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial Mathematics, 1 edition, (1987).

58. Chaudhuri, M., Heinrich, M., Holt, C., Singh, J., Rothberg, E., and Hennessy, J. *Computers, IEEE Transactions on* **52**(7), 862 – 880 jul (2003).

59. Protic, J., Tomasevic, M., and Milutinovic, V. *Parallel Distributed Technology: Systems Applications, IEEE* **4**(2), 63 –71 summer (1996).

60. Itzkovitz, A., Niv, N., and Schuster, A. *Journal of Systems and Software* **55**(1), 19 – 32 (2000).

61. Bohm, A. and Kanne, C.-C. *Information Systems* **36**(3), 565 – 578 (2011). ¡ce:title¿Special Issue on WISE 2009 - Web Information Systems Engineering¡/ce:title¿.

62. Kirk, D. B. and Hwu, W. W. *Programming massively parallel processors: A hands on approach.* Morgan Kaufmann, (2010).

63. Nvidia corporation. *NVIDIA GeForce GTX 200 GPU Architectural Overview: Second-Generation Unified GPU Architecture for Visual Computing*, (2008).

64. NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, version 3.0 edition, February (2010).

65. Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, 35–46 (ACM, New York, NY, USA, 2011).

66. Bouchez, F. Technical report, Indian Institute of science, Bangalore, (2010).

67. Accelereyes. November (2011).

68. Iveson, S. M., Litster, J. D., Hapgood, K., and Ennis, B. J. *Powder Technology* **117**(12), 3 – 39 (2001).

69. Iveson, S. M. *Chemical Engineering Science* **56**(6), 2215 – 2220 (2001).

70. Iveson, S. M. and Litster, J. D. *AIChE Journal* **44**(7), 1510–1518 (1998).

71. Tu, W.-D., Ingram, A., Seville, J., and Hsiau, S.-S. *Chemical Engineering Journal* **145**(3), 505 – 513 (2009).

72. Kumar, J. *Numerical approximations of population balance equations in particulate systems.* PhD thesis, (2006).

73. Kumar, J., Peglow, M., Warnecke, G., and Heinrich, S. *Powder Technology* **182**(1), 81 – 104 (2008).

74. Kumar, S. and Ramkrishna, D. *Chemical Engineering Science* **51**(8), 1333 – 1342 (1996).

75. Kumar, S. and Ramkrishna, D. *Chemical Engineering Science* **51**(8), 1311 – 1332 (1996).

76. Chaudhury, A., Kapadia, A., V., A., Barrasso, D., and Ramachandran, R. *Computers and Chemical Engineering* , Manuscript submitted. (2012).

77. Barrasso, D. and Ramachandran, R. *Chemical Engineering Science* **80**(0), 380 – 392 (2012).

78. Immanuel, C. D. and Doyle III, F. J. *Chemical Engineering Science* **58**(16), 3681 – 3698 (2003).

79. Flynn, M. J. *IEEE Trans. Comput.* **21**(9), 948–960 September (1972).

80. Klockner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. *Parallel Computing* **911**, 1–24 (2011).

81. Refianti, R., Refianti, R., and Hasta, D. In *International Journal of Advanced Computer Science and Applications (IJACSA)*, volume 2, 99–107, (2011).

82. Haines, M. D. *Distributed runtime support for task nd data management.* PhD thesis, Colorado State University, (1993).

83. Haveraaen, M. *SCIENTIFIC PROGRAMMING* **8**, 231–246 (2000).

84. Zhang, Y., Mueller, F., Cui, X., and Potok, T. *Journal of Parallel and Distributed Computing* **71**(2), 211–224 (2011).

85. Martin, M. M. K., Hill, M. D., and Sorin, D. J. Technical report, Duke University, Department of ECE, August (2011).

86. Brightwell, R., Camp, W., Cole, B., Debenedictis, E., Leland, R., Tomkins, J., and Maccabe, A. B. *Concurrency and Computation: Practice and Experience* **17**(10), 1217 – 1316 (2005).

87. Akl, S. G. *Journal of Supercomputing* **29**, 89–111 (2001).

88. Gustafson, J. L., Montry, G. R., Benner, R. E., and Gear, C. W. *SIAM Journal on Scientific and Statistical Computing* **9**, 609–638 (1988).

89. Amdahl, G. M. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), 483–485 (ACM, New York, NY, USA, 1967).

90. Wu, X. and Li, W. *Journal of Systems Architecture* **44**(34), 189 – 205 (1998).

91. Gupta, A., Gupta, A., and Kumar, V. Technical report, Department of Computer Science, University of Minnesota, (1993).

92. Heath, M. T. Lecture at University of Illinois at Urbana-Champaign.

Vita

Anuj Varghese Prakash

2013	M. S. in Chemical and Biochemical Engineering, Rutgers University
2009	B. Tech. in Biotechnology and Biochemical Engineering from Kerala University, India
2005	Graduated from M.E.S Indian School, Qatar.

Publications

1. A quantitative assessment of the influence of primary particle size polydispersity on granule inhomogeneity. Rohit Ramachandran, Mansoor A. Ansari, Anwesha Chaudhury, Avi Kapadia, Anuj V. Prakash, Frantisek Stepanek. Chemical Engineering Science. March 26, 2012

2. Parallel simulation of population balance model-based particulate processes using multi-core CPUs and GPUs. Anuj V. Prakash, Anwesha Chaudhury, Rohit Ramachandran. Chemical Engineering Science (Under Review)

3. Simulation of population balance model-based particulate processes on a distributed computing system. Anuj V. Prakash, Anwesha Chaudhury, Dana Barrasso, Rohit Ramachandran. Journal of Parallel and Distributed Computing (Under Review)

4. An Extended Cell-average Technique for Multi-Dimensional Population Balance Models describing Aggregation and Breakage. Anwesha Chaudhury, Avi Kapadia, Anuj V. Prakash, Dana Barrasso, Rohit Ramachandran. Computers and Chemical Engineering (Under Review)