Hindawi Publishing Corporation
Modelling and Simulation in Engineering
Volume 2013, Article ID 475478, 16 pages
http://dx.doi.org/10.1155/2013/475478

Research Article

Parallel Simulation of Population Balance Model-Based Particulate Processes Using Multicore CPUs and GPUs

Anuj V. Prakash, Anwesha Chaudhury, and Rohit Ramachandran

Department of Chemical and Biochemical Engineering, Rutgers, The State University of New Jersey, Piscataway, NJ 08854, USA

Correspondence should be addressed to Rohit Ramachandran; rohit.r@rutgers.edu

Received 15 October 2012; Revised 12 February 2013; Accepted 19 February 2013

Academic Editor: Weizhong Dai

Computer-aided modeling and simulation are a crucial step in developing, integrating, and optimizing unit operations and subsequently the entire processes in the chemical/pharmaceutical industry. This study details two methods of reducing the computational time to solve complex process models, namely, the population balance model which given the source terms can be very computationally intensive. Population balance models are also widely used to describe the time evolutions and distributions of many particulate processes, and its efficient and quick simulation would be very beneficial. The first method illustrates utilization of MATLAB's Parallel Computing Toolbox (PCT) and the second method makes use of another toolbox, JACKET, to speed up computations on the CPU and GPU, respectively. Results indicate significant reduction in computational time for the same accuracy using multicore CPUs. Many-core platforms such as GPUs are also promising towards computational time reduction for larger problems despite the limitations of lower clock speed and device memory. This lends credence to the use of highfidelity models (in place of reduced order models) for control and optimization of particulate processes.

1. Introduction

Modeling and simulation are powerful tools universally employed in designing, analyzing, and controlling particulate processes. These particulate processes such as crystallization, granulation, milling, and polymerization are some of the major unit operations carried out in the manufacture of bulk commercial products like pharmaceuticals, detergents, fertilizers, and polymers. Research work focusing on the modeling and simulation of these particulate processes, specifically those involving granular materials, has been growing at a steady pace over the last few decades [1–3]. This is a significant achievement in itself, considering the fact that these systems are inherently dynamic in behavior and are driven by complex microscale phenomena [4]. Although the underlying mechanisms of such processes are yet to be thoroughly grasped, granulation, a particle design process, is one such area where substantial progress has been made over the years [5]. The approaches for modeling such systems are as numerous as they are varied: Discrete Element Modeling (DEM) [6], Population Balance Modeling (PBM) [3, 7–13], hybrid models by combining PBM with DEM [14], PBM with Volume of Fluid (VoF) methods [15], and PBM with Computational Fluid Dynamics (CFD) [16], to name a few. Of the aforementioned, the most widely used are the DEM and PBM methods. Population Balance (PB) models are more suited to simulate a very large number of particles over lengthy time periods due to the semimechanistic approach (compared to more mechanistic approaches such as DEM and VoF) it utilizes to describe the dynamics of granulation processes [17]. Because of these advantages, PBM offers a very efficient manner of developing a comprehensive model of a granulation process, which can be simulated within a realistic time frame to be further used in control and optimization [18], since it provides a convenient mathematical framework whereby the detail of the model is user specific and depends on the kernel formulations specified by the user [7].

However, as with all model-based simulations, the utility of the PBM technique depends on the computational expenses it incurs in terms of run time and hardware resources. In addition to increased numerical accuracy, the need for speed has always been a demand for the scientific community to handle larger and more complex problems as well as the industrial community who use process models to study their processes *in silico*. This is demonstrable by the fact that even in a high-level language environment like MATLAB, the computational load increases almost polynomially on increasing the dimensionality of a system, leading to longer run times. 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 [19]. While MATLAB excels on the "ease of programmability" and "portability" fronts, it has been found to be lacking in the "performance" department [20]. 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 distributed process such as granulation typically has several nested for loops and multiple operations over large data sets that are executed "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 (Central Processing Units) 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 multicore/multiprocessor CPUs. Furthermore, due to limitations on the power density that can be supplied, attainable peak CPU clock frequency is restricted (6 GHz for an Intel Core i5 [21]).

By parallelizing an existing code, the programmer is able to circumvent the restriction of running a code sequentially on one core and in addition exploit other massively parallel processors like the GPU (Graphics Processing Unit) [22]. Since 2006, MATLAB comes standard with a toolbox for this purpose called the Parallel Computing Toolbox (PCT), although for GPU computation, the toolbox from "Accelereyes inc." named "JACKET" is chosen as it clearly outperforms MATLAB's built-in capabilities [23]. There has been some recent work on the parallelization of PBM simulations prior to this study. Gunawan et al. formulated an efficient way of parallelizing High-Resolution Finite Volume- (HRFV-) solved PBEs 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 [24]. Their strategy enabled efficient load distribution and resulted in a near linear speedup. More recently, Ganesan and Tobiska [25] 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. Both papers outlined innovative techniques for parallelization of a PBM for multiple processing units. This paper intends to provide a means of significantly mitigating the handicaps of PB simulations by demonstrating how parallel processing capabilities of multicore CPUs, as well as GPUs, can be harnessed within a high-level language environment like MATLAB by using both in-built functionalities as well as third-party toolboxes. The focus here will not be on developing specialized parallel codes from the ground up that will eventually be application and/or hardware limited, but rather provide the modelling and research community at large with the aforesaid tools to parallelize their codes with minimum effort. GPU computing has also been applied to mixing processes described by DEM as seen in the work of Radeke et al. [26] thus confirming its usefulness in mitigating computational times of complex process models.

2. Background

2.1. Population Balance Models. Population balance models have traditionally been one dimensional described by a single intrinsic property such as particle size [27]. A general form of the population balance equation (1) highlighting the temporal variation of the distribution of one or more intrinsic properties is given as follows [28]:

$$\frac{\partial F}{\partial t}(\mathbf{x},t)+\frac{\partial}{\partial \mathbf{x}}\left[F(\mathbf{x},t)\frac{d\mathbf{x}}{dt}\right]=\mathfrak{R}_{\text{formation}}(\mathbf{x},t)-\mathfrak{R}_{\text{depletion}}(\mathbf{x},t), \tag{1}$$

where F is the particle number distribution and $\mathbf{x}$ is the vector of internal coordinates, which are of interest for studying the process. $\mathfrak{R}_{\text{formation}}$ and $\mathfrak{R}_{\text{depletion}}$ represent the net formation and depletion rates of particles occurring from all discrete granulation mechanisms such as aggregation, nucleation, and breakage.

However, dependence on particle size only was found to be inadequate in characterizing the variability in granulation behavior and thereafter, other factors like granule porosity were also found to exert a dominating effect on the process [29, 30]. Consequently, 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 the current research on granulation [11], which involves model development within a three-dimensional population balance framework. Verkoeijen et al. [31] had previously described an efficient way of implementing such a framework by expressing the intrinsic properties of granules—that is, the volume of solids s, volume of liquid l, and volume of gas g—as a vector in volume space with three coordinates, that is, s, l, and g. Particle internal coordinates (2) are now represented as

$$\mathbf{x}=\left[s\ \ l\ \ g\right], \tag{2}$$

where each of these three coordinates (s,l,g) comprises unique distributions of phase volumes (solid, liquid, or gas) of all particles belonging to a predefined volume class, and can therefore be represented as three separate discretized domains or "grids," containing the distributions. These grids are composed of "bins" that denote the volume classes to the particles present in the population. The first bin in the solid volume grid represents the particles that have the least solid content. This way, the individual solid volumes of the particles can be represented by allocating them in the corresponding bins. The same principles apply to the other two phase fraction grids, liquid (l) and gas (g). From now on, the term "grid size" will be used to refer to the total number of bins in a grid. This 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 [7]. This 3-dimensional model can now describe changes in the volume distribution of particle volume with respect to time [3], as follows:

$$\begin{aligned}\frac{\partial}{\partial t}F(s,l,g,t)&+\frac{\partial}{\partial g}\left[F(s,l,g,t)\frac{dg}{dt}\right]\\&+\frac{\partial}{\partial s}\left[F(s,l,g,t)\frac{ds}{dt}\right]+\frac{\partial}{\partial l}\left[F(s,l,g,t)\frac{dl}{dt}\right]\\&=\Re_{\text{aggregation}}+\Re_{\text{breakage}}+\Re_{\text{nucleation}},\end{aligned}\tag{3}$$

where $F(s,l,g,t)$ represents the population density function such that $F(s,l,g,t)ds\,dl\,dg$ is 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 rewetting 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. On the right-hand side, the $\Re_{\text{breakage}}$ term comprises a breakage kernel and a breakage function; $\Re_{\text{nucleation}}$ accounts for the rate of nucleation of new particles. $\Re_{\text{nucleation}}$ and $\Re_{\text{breakage}}$ are not utilized in this study and therefore not described in more detail. The authors would like to direct the readers to Poon et al. [8] and Ramachandran et al. [3] for their descriptions. $\Re_{\text{aggregation}}$ (4)–(6) takes into account the formation/depletion of granules due to aggregation, for which the terms have been defined in the literature [32] as

$$\Re_{\text{agg}}(s,l,g,t)=\Re_{\text{agg}}{}^{\text{formation}}-\Re_{\text{agg}}{}^{\text{depletion}},\tag{4}$$

$$\begin{aligned}\Re_{\text{agg}}{}^{\text{formation}}\\&=\frac{1}{2}\int_{s_{\text{nuc}}}^{s-s_{\text{nuc}}}\int_0^{l_{\max}}\int_0^{g_{\max}}\beta\left(s',s-s',l',l-l',g',g-g'\right)\\&\quad\times F\left(s',l',g',t\right)\\&\quad\times F\left(s',s-s',l',l-l',g',g-g',t\right)ds'\,dl'\,dg',\end{aligned}\tag{5}$$

$$\begin{aligned}\Re_{\text{agg}}{}^{\text{depletion}}\\&=F(s,l,g,t)\\&\quad\times\int_{s_{\text{nuc}}}^{s-s_{\text{nuc}}}\int_0^{l_{\max}}\int_0^{g_{\max}}\beta\left(s',s-s',l',l-l',g',g-g'\right)\\&\quad\times F\left(s',l',g',t\right)ds'\,dl'\,dg',\end{aligned}\tag{6}$$

where s_{nuc} is the solid volume of nuclei and $\beta(s',s-s',l',l-l',g',g-g')$ is the size-dependent aggregation kernel that signifies the rate constant for aggregation of two granules of internal coordinates (s',l',g') and $(s-s',l-l',g-g')$.

2.2. Parallel Computing for the CPU and GPU.

To speed up calculations, a single problem or "task" is split into multiple subtasks, which are executed simultaneously on multiple processors. This method of program execution is termed "parallel computing" or "parallel processing" as opposed to "serial" or "sequential" execution, and developing scripts that leverage this style of execution is called "parallel programming" [33]. In the current generation of multicore processors, there are multiple independent processing units called "cores" which carry out a set of instructions. A single processor can consist of many such cores, with each core capable of executing an instruction set. Although a "core" refers to the physical component providing parallelism, in general it can also mean a thread (a piece of software) or a processor or even a machine (on a network) executing a stream of instructions [34]. For instance, the Intel Core i7-2600K processor has four physical cores, each with two threads raising the number of (logical) cores to eight [35]. The fundamental concept behind parallel programming is that n cores/processors should provide a peak speedup of n times over just one core/processor. However, such gains in simulation time are at best theoretical, simply because the time needed for data transfer and synchronization to, from, and between cores negate's any benefit in speedup [36]. Depending on the hardware architecture of the 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 [37, 38]. The most widely used approaches are task parallelism, data parallelism, and distributed memory/message passing model. An elaborate explanation of each model is beyond the scope of this paper, so simple definitions and possible modes of implementation are provided instead [39, 40].

(i) Task parallelism is achieved by assigning each task (or subtask) to a unique core or thread (threads model) and finally splitting or combining the data stream at the end. Implementation: POSIX threads, Open MP.

(ii) Data parallelism involves dividing a large amount of data into sections across cores, each of which is then operated upon by the same task within each core. Implementation: Fortran 90 and 95.

(iii) In message passing, each subtask 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).

2.2.1. CPU Architecture and Parallel Programming. Computing architecture can be classified, based on Flynn's scheme [41], 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 falls in the MIMD category but utilizes SISD (single instruction, single data) processing units at the lowest level [42]. There are three common approaches to implement parallel execution on these system: SIMD (SSE) instructions operating on multiple data sets in parallel with a single instruction stream; simultaneous multithreading (SMT), popularly called "hyperthreading"; or as is now generally preferred through custom libraries or "toolboxes" like MATLAB's Parallel Computing Toolbox (PCT). PCT enables the developer to take advantage of multicore processors, GPUs, and computer clusters by making available high-level constructs such as parallel for loops `parfor`, specialized arrays (distributed and codistributed arrays), and preparallelized numerical algorithms. This allows the programmer to focus on building the algorithm and not on micromanaging parallel communication between cores, which are taken care of by MATLAB behind the scenes. These parallel programming constructs will function in the same way, independent of the underlying hardware component being used, whether it is a multicore desktop via PCT, or on a network of computers (computer cluster) via PCT with the MATLAB Distributed Computing Server (MDCS) package [43].

In MATLAB, each task is handled by an independent instance of MATLAB called a *worker* or *lab* that runs as a separate system process. Communication to, from, and between these workers is handled by the *client* instance of MATLAB. These labs 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 multithreading that is built into MATLAB, there are explicit methods of parallelism available to the developer as well [44]. The `parfor` keyword is perhaps the easiest way to achieve parallelism in an existing code with little modification. Just replacing the (preferably) outermost `for` with a `parfor` in a for-loop results in substantial speedup, sometimes proportional to the number of cores depending on the problem. The job of distributing iterations and collecting end results is handled by MATLAB without any requirement for explicit commands from the programmer. But this gain is soon lost when the number of labs exceeds the number of cores, since communication overhead is always higher between threads than cores [45]. Another explicit approach to parallelize a code is by using the `SPMD` keyword. `SPMD` (Single Program Multiple Data) is a high-level construct that can be built upon a combination of the aforementioned task, data, and message passing types of parallelism. Each MATLAB worker is assigned the same program, which operates on different arrays or different sections of a very large data array, hence the term "Single Program Multiple Data." Furthermore, if data exchange and synchronization between the workers is desired, functions based on the Message Passing Interface (MPI) library [46] like *LabSend()* and *LabReceive()* are available in conjunction with the `SPMD` keyword. Other forms of explicit parallelism include distributed and codistributed arrays, which will not be considered in this paper. For any of the constructs described before are to be implemented, a `matlabpool` open command must be issued beforehand in order for the client session to establish a connection with available workers. For further information on PCT constructs and their implementation we refer to the appropriate section of the MATLAB manual [43].

2.2.2. GPU Architecture and Parallel Programming. The GPU is an excellent example of the SIMD design paradigm. A GPU is organized as an array of many cores, or as NVIDIA describes, "streaming multiprocessors" (SMs). Each SM has a certain number of ALU (Arithmetic and Logic) units called streaming processors (SPs), which share a common control logic and instruction cache. While the CPU design paradigm boasts excellent performance in sequential operations, the presence of a complex control logic and large cache memory limits the maximum speed achievable in gigaflops [47]. The GPU control logic systems, on the other hand, are not as bulky, with the GPU themselves fabricated as relatively wide SIMD vectors, increasing their parallel processing capacity. Owing to their architecture, GPUs are specialized for data-parallel calculations, unlike MIMD-based platforms like the Core i7 which are 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/ALUs, each of nearly 1.3 GHz frequency. Each SM has 1024 threads, bringing the total to 30,720 threads within a single GPU [48].

To program these massively parallel architectures, NVIDIA developed the Compute Unified Device Architecture (CUDA), which was released in 2006, permitting high-level programmability within the C language [49]. 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 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 [50, 51]. 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 [52]. 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 subtasks, which could then be executed independently of each other on separate cores. This kind of problem is termed as "embarrassingly parallel" [36]. In reality, most problems lie somewhere between this extreme and the "annoyingly sequential" extreme.

3. Model Parallelization Strategy

Parallelization of the code with respect to both the CPU and GPU involved the SPMD approach (outlined in the previous section) combining both data- and task-parallel styles of programming. 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 that is required by particulate processes such as granulation) typically comprises 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 [53]. 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 the purpose of this study, a 3D population balance model based on (3) was developed with the following simplifications:

(i) aggregation as the only source term, eliminating breakage and nucleation terms ($s_{\text{nuc}} = 0$);

(ii) "growth terms": drying/rewetting, layering, and consolidation are neglected;

(iii) an empirical aggregation kernel proposed by Madec et al. [54] is used,

yielding the following PBE (7). Please see Appendix A for details of the aggregation kernel used and Appendix D for the numerical solution of the PBM based on a hierarchical two-tiered algorithm proposed by Immanuel and Doyle III [55]:

$$\frac{\partial}{\partial t} F(s,l,g,t) = \mathfrak{R}_{\text{aggregation}}. \tag{7}$$

This was done to highlight the improvement in simulation speed achieved by parallelizing only the formation/depletion code blocks of a PBM script, which tend to be the most computationally intensive. It was observed that removing the formation and depletion terms associated with aggregation from a PBM code (that considered all mechanisms) resulted in only a 20% faster simulation time, proving that aggregation is indeed the primary computational bottleneck. This is due to the presence of multiple "nested for loops," prominently those that account for the integral equations (5) and (6) running sequentially on a single CPU core. In other words, broadening the range of each loop index causes individual iterations to run slower. Additionally, there are numerous such for-loops and sequential sections of code performing calculations independently of each other that can be parallelized. Increasing the number of bins/grids in each dimension with respect to s, l, and g, while raising the dimensionality of the system, also slows down the code execution considerably. This is also termed the curse of dimensionality phenomenon. Although it is preferred to use a higher grid size for an accurate representation of the system, the aforementioned shortcomings curb the degree of flexibility available to a researcher and/or industrial practitioner. Therefore, there is much potential for speedup in parallelizing these loops to run simultaneously on all cores/processors present on the device.

Following the procedure just described, execution of the aggregation kernel can be parallelized by "slicing" the outermost loop:

$$\begin{aligned}\mathfrak{R}_{\text{agg}}^{\text{formation}} = & \int_0^{\text{upper}_1}\int_0^{l_{\max}}\int_0^{g_{\max}} \text{form}(s,l,g) \\ & + \int_{\text{upper}_1+1}^{\text{upper}_2}\int_0^{l_{\max}}\int_0^{g_{\max}} \text{form}(s,l,g) \\ & + \int_{\text{upper}_2+1}^{\text{upper}_3}\int_0^{l_{\max}}\int_0^{g_{\max}} \text{form}(s,l,g)\cdots \\ & + \int_{\text{upper}_n+1}^{s-s_{\text{nuc}}}\int_0^{l_{\max}}\int_0^{g_{\max}} \text{form}(s,l,g),\end{aligned} \tag{8}$$

where

$$\text{upper}_n = \frac{s - s_{\text{nuc}}}{\texttt{numlabs}} \times \texttt{labindex}. \tag{9}$$

JACKET's `gfor` employs an algorithm similar to the one above 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 approach just described, loop slicing, allows greater control of data distribution across workers, reducing the demand for system resources over time in a "smoothed-out" fashion [56]. Besides the data-parallel approach, another, more straightforward *divide-and-conquer* method involves task parallelism. Implementations of task parallelism are generally done through the fork-join model, described in Refianti et al. [57], 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 (meaning fewer, but larger tasks) parallelism with consecutive, but independent sections of the code being mapped onto different threads and task scheduling done at the time of compilation, that is, 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 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 [58].

Both the task- and data-parallel approaches follow the same algorithm: at the start of the simulation, only the MATLAB client instance is actively processing code sequentially. On seeing an `SPMD` keyword, the code then forks off function calls onto idle workers in a parallel manner. With every worker active, execution of the allocated serial tasks now begins asynchronously. After all the workers have completed their respective tasks, they return their results to the client instance as `Composite` types, which can then be cast back to regular CPU `single` or `double` types and subsequently rejoined. To sum up, 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 aforementioned approaches for parallelism as appropriate; and finally optimizing for minimal variable transfer overhead.

4. Results and Discussion

4.1. Comparing CPU-`for`, GPU-`for`, and GPU-`gfor` Execution. The first set of simulations was conducted to compare the speed gains obtained by running the case with only aggregation PBM code, based on (7), first on a GPU and then 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 (termed the "gpu-`for`" version) and in the other, termed the `gfor` version, JACKET's `gfor` constructs were used instead. The CPU version was left unparallelized, that is, with regular `for`-loops, to execute sequentially on a single MATLAB worker. Simulation was carried out on a machine with a Core2Quad Q6600 processor (2.4 GHz clock, 4 cores, no threads), 4 GB of RAM (2 GB × 2 sticks), and an NVIDIA GeForce GTX 280 GPU (240 CUDA cores, 1296 MHz processor clock, 1 GB memory). Results from the simulation of each of these three cases were first validated by comparing bulk property plots of total number of particles versus time, total volume versus time, and average diameter versus time after the final time step to verify uniformity. This was followed by plotting the time taken to simulate each case versus grid size and then the speedup ratio versus grid size. The ratios were calculated as

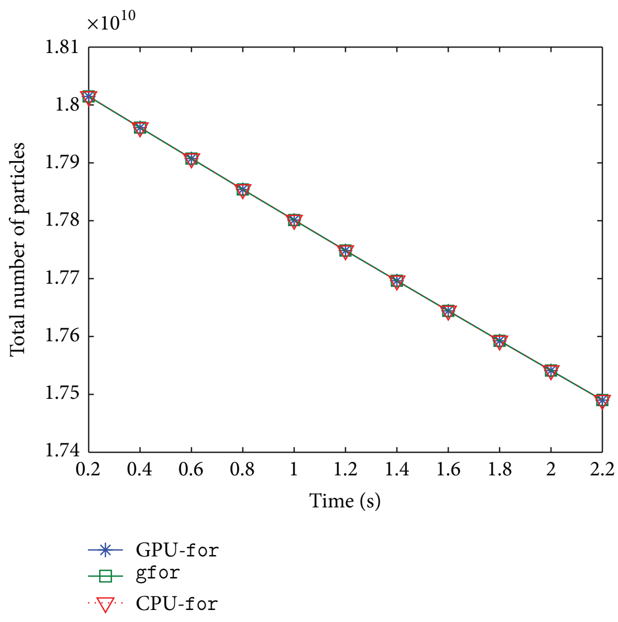
$$\text{Ratio} = \frac{\text{single CPU time}}{\texttt{gfor}\ \text{time}} \qquad (10)$$

or

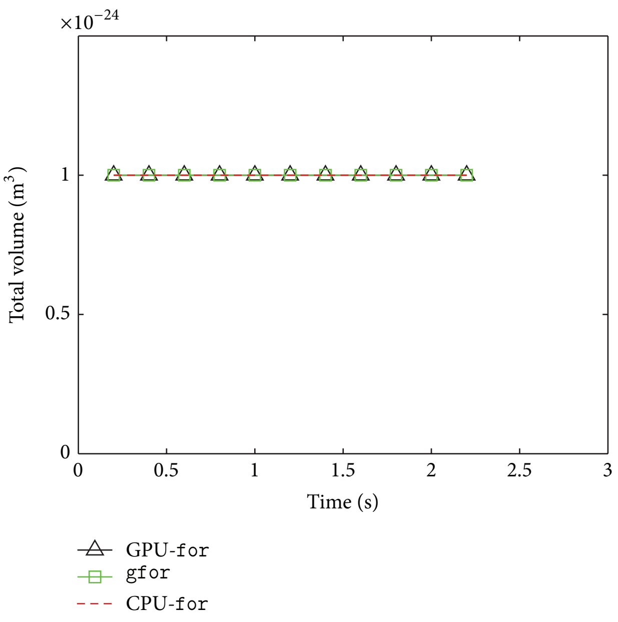
$$\text{Ratio} = \frac{\text{GPU}\ \ \texttt{fortime}}{\texttt{gfor}\ \text{time}}. \qquad (11)$$

From the curves depicting temporal evolution of granule properties (Figure 1), 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 of particles (Figure 1(a)) decreases at a constant rate due to aggregation, wherein the collision (and therefore, depletion) of two particles leads to the formation of a new one by coalescence [8]. An analysis of the total volume plot, Figure 1(b), predictably reveals constant value lines considering the fact that total mass/volume is conserved in the system; that is, 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 1(c), shows a proportional increase in the size of granules over time. Calculations for these bulk properties are the same for all simulation cases and can be found in Appendix E.

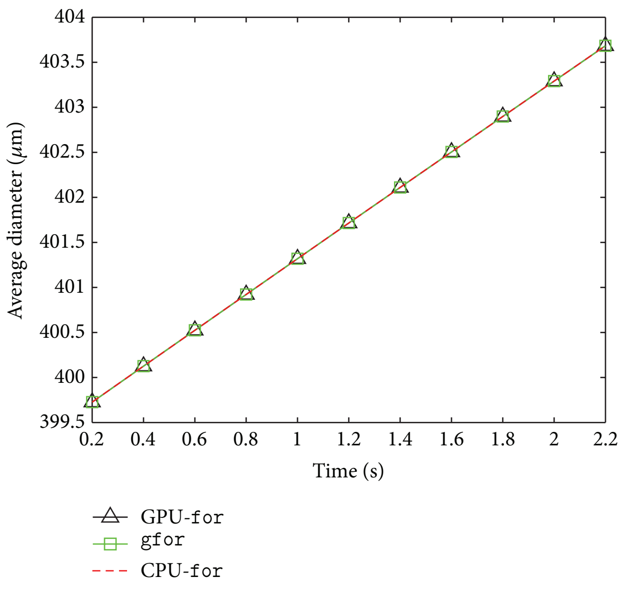
The simulation time versus grid size curves, Figure 2(a), show the single-worker CPU version of the code to be much faster than its GPU counterparts, with the slowest of the set being the code with GPU-`for`-loops, followed by the `gfor` loop version. It must be noted that the GPU is a stand-alone device and does not share its memory with the host (CPU) or provide a means for virtual memory addressing. In other words, data will not be communicated automatically between the host and the device memories, but rather explicitly invoked. This causes severe memory transfer overheads each time a variable is copied to and from the GPU across the PCI-E bus [59], 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 2(c), with the ratio calculated by (11). Although preliminary results 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 2(b). The steady increase in the ratio (10) 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 2(a) and in Figure 2(b) is an anomaly shown to be reproducible even after initiating the simulation at various grid size values and is probably due to an initial memory overhead during the "warm-up" of the CPU before commencing code execution. In addition to the aforementioned hardware limitations of the GPU, JACKET's execution of a script is not transparent to the programmer, and capabilities in terms of benchmarking, assigning tasks to specific thread blocks, and controlling memory access patterns are nonexistent. Future work will involve building a GPU-efficient code from the ground up,



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



(b) Evolution of total volume of particles over time

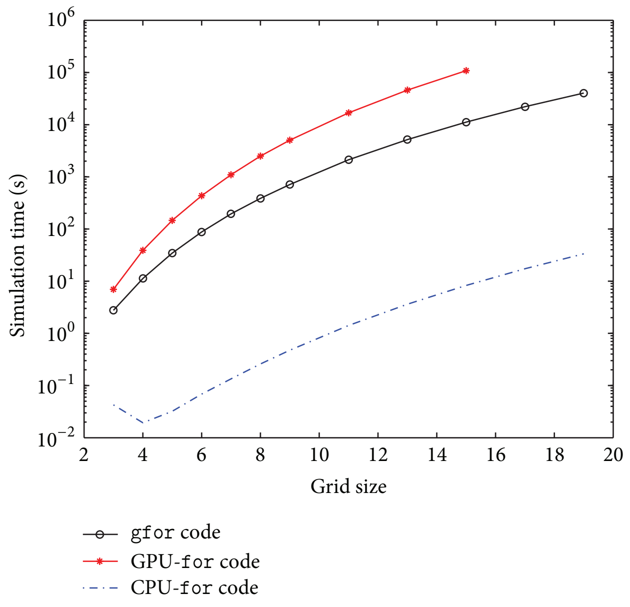


(c) Evolution of average diameter of a particle over time

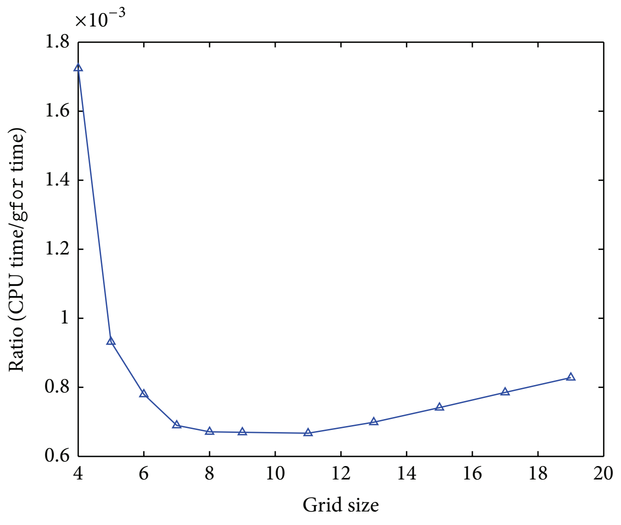
Figure 1: Comparison of temporal evolution of granule physical properties simulated using `gfor`, GPU-`for`, and CPU-`for`.

which will lend itself better to parallelization in conjunction with constructs like `gfor`.

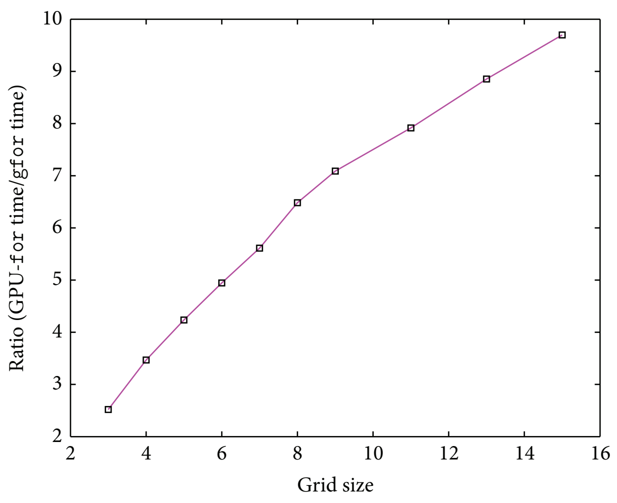
4.2. Comparing Single CPU and SPMD Execution. Next, a comparison of the simulation times for the PBM code to run on a single worker sequentially and then on multiple workers was done, followed by plotting the speedup gained. Prior to execution, the code was "streamlined" to efficiently search for and 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 [52]. 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) 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 the efficiency of



(a) Semilog 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

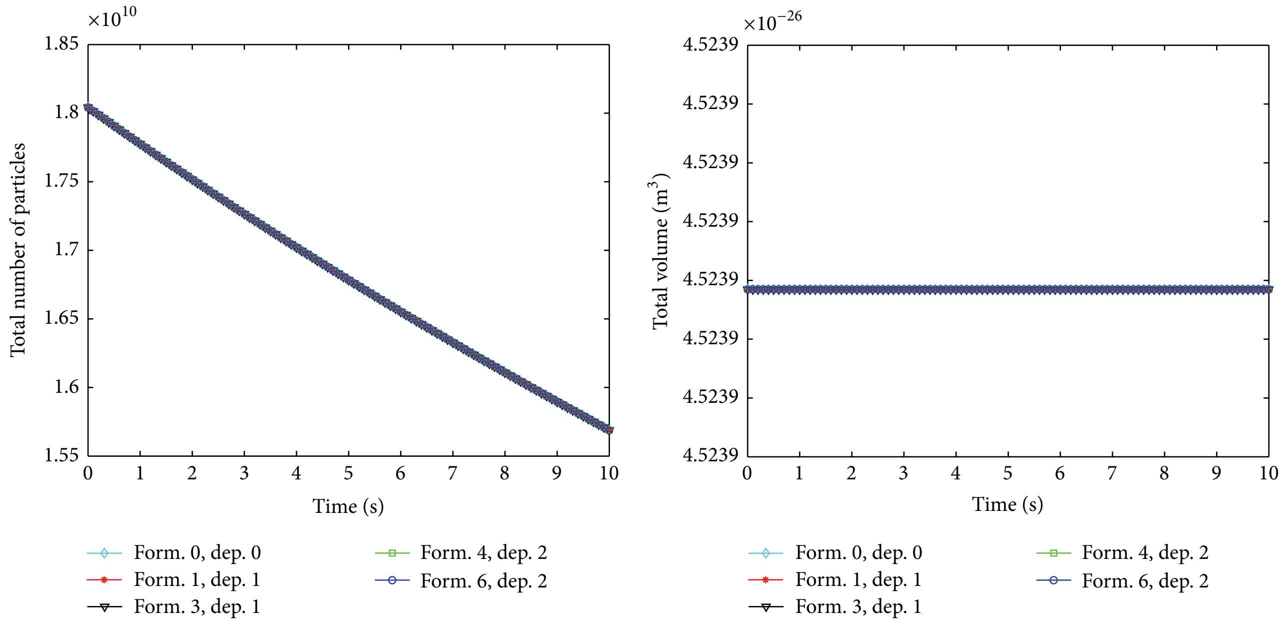
Figure 2: Comparison of simulation times and speedup ratios of PBM code incorporating `gfor`, GPU-`for`, and CPU-`for`.

simulation (refer Table 1). Although formation is the primary computational bottleneck requiring *loop slicing*, the initial test runs in conjunction with MATLAB's Profiler tool affirmed that it was also necessary for depletion to execute *at least* one worker 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.

Within these combinations, the ones yielding 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 worker); 1 formation, 1 depletion (2 workers); 3 formations, 1 depletion (4 workers); and 6 formations, 2 depletions (8 workers). As done previously, the plots for

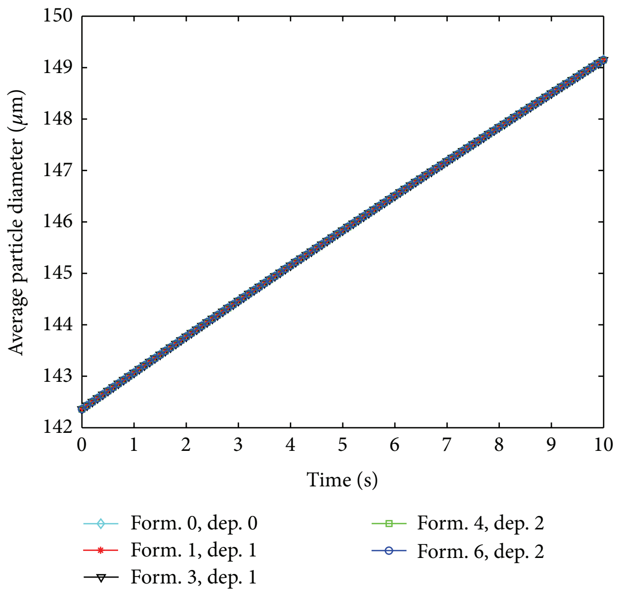
Table 1: Loop slicing combinations.

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



(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

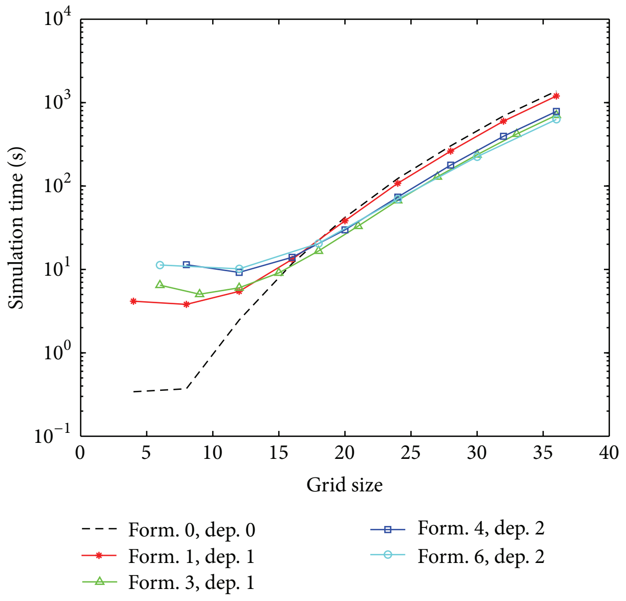
FIGURE 3: Comparison of temporal evolution of granule physical properties simulated for different worker pool classes, grid size = 36.

granule physical properties, Figures 3(a)–3(c), were examined to ensure validity and numerical precision of the results. Having confirmed that, the simulation times, the parallel speedup, and efficiency curves were plotted for the five worker pool classes selected (Figures 4(a)–4(c)).

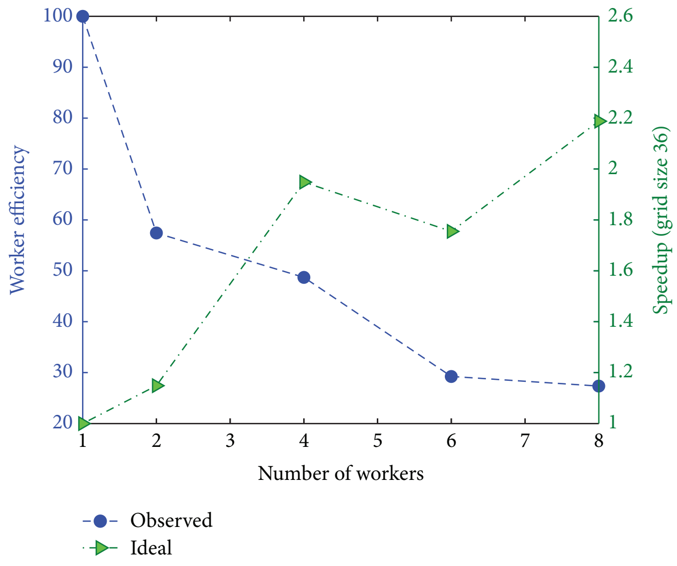
The speedup factor and parallel efficiency were calculated as given in Wilkinson and Allen [36]:

$$\text{Speedup } S(n) = \frac{\text{Execution time on a single worker}}{\text{Execution time on } n \text{ workers}},$$
$$\text{Parallel Efficiency } E = \frac{S(n)}{n} \times 100. \tag{12}$$

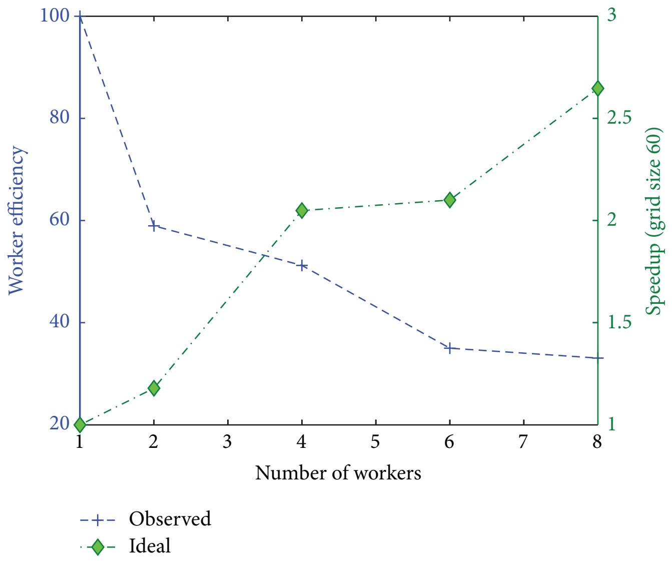
Simply put, the speedup factor directly quantifies the gain in performance of multiprocessor system over a single-processor one. As observed in Figure 4(b), the maximum speedup achieved with 8 workers was 2.2 times, leading to an average per worker efficiency of 27.35. Parallel efficiency is a measure of computational resource usage, with lower values implying lower utilization and higher values implying higher 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 4(c). Furthermore, it was also



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



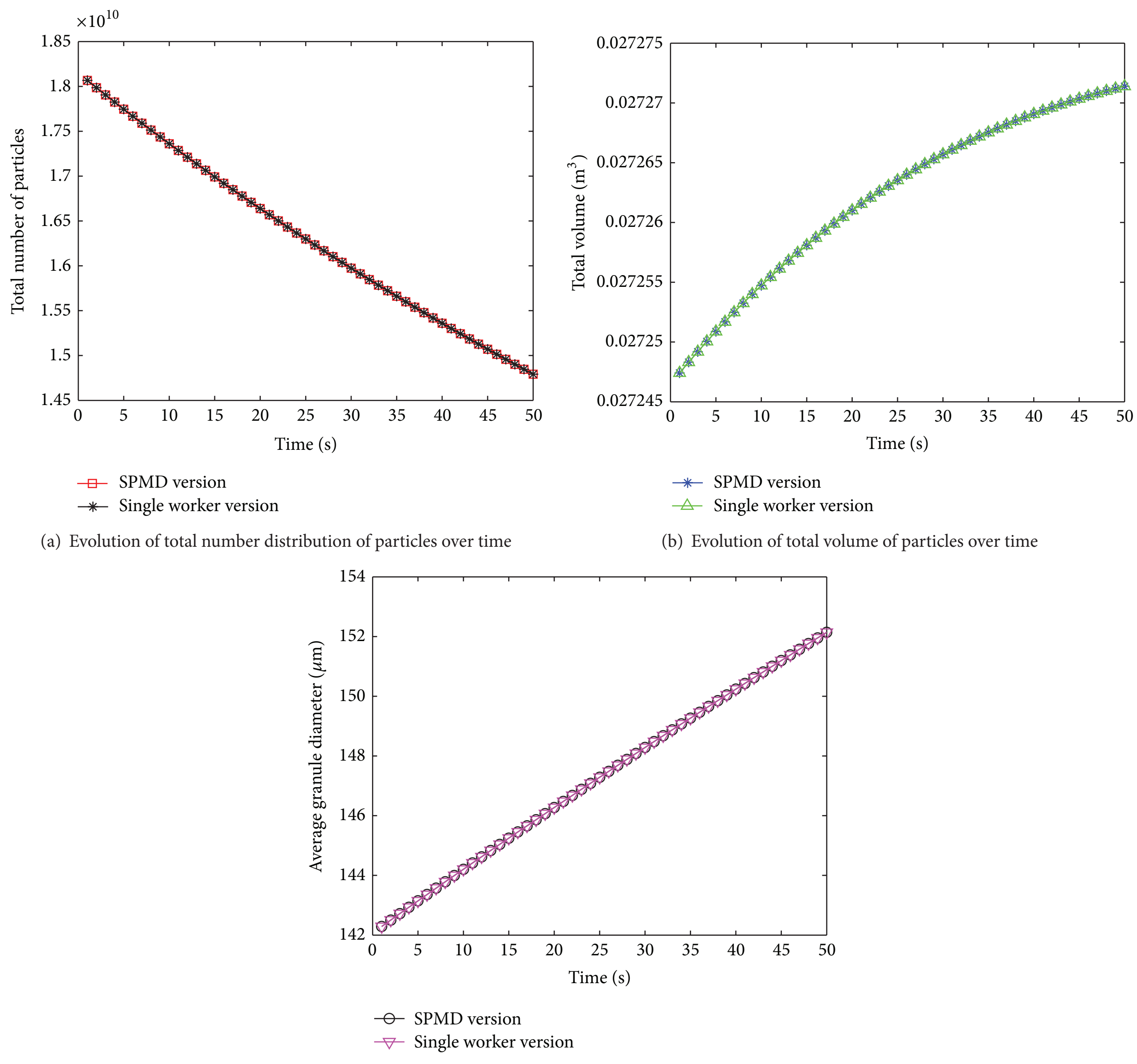
(b) Speedup and efficiency obtained for a grid size of 36



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

Figure 4: Plots of simulation times and obtained speedup of the PB code incorporating the SPMD construct.

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, Figures 4(a) and 4(b)). Currently, we have restricted ourselves to a grid size of 60 due to MATLAB's limit on the maximum possible array size (proportional to available system RAM), but future work will involve working around these memory limitations using distributed data types and employing constructs like `labSend` and `labReceive` for better memory read/write patterns. 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



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

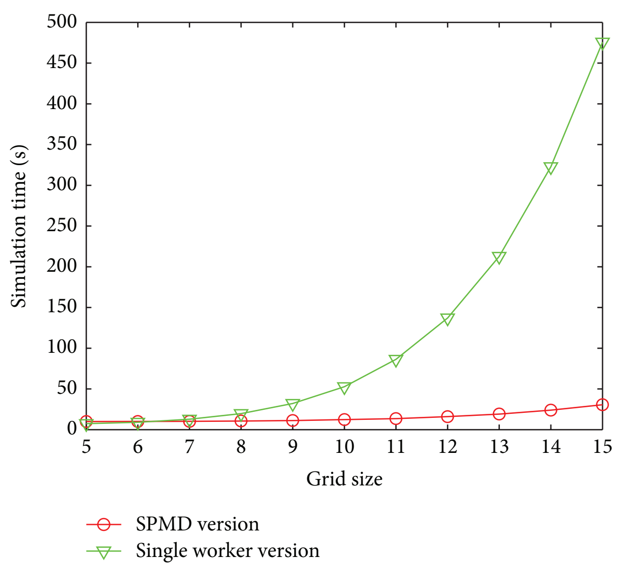
(b) Evolution of total volume of particles over time

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

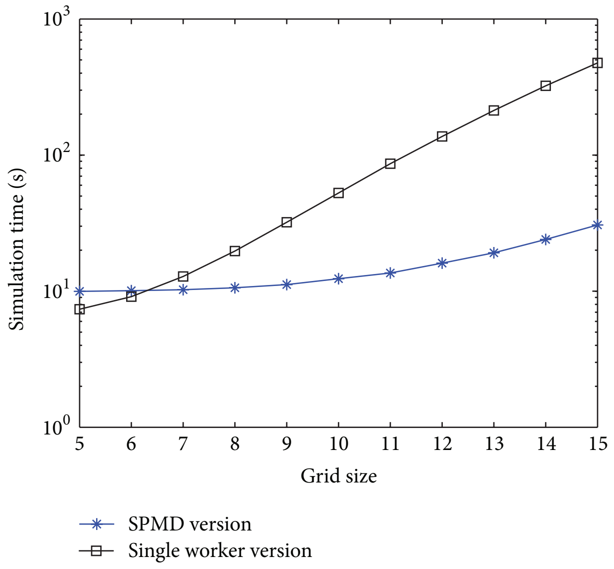
FIGURE 5: Comparison of temporal evolution of granule physical properties for a sequential and parallel PBM code, grid size = 15.

multiprocessing architecture like the INTEL Core i7 [60]; and memory conflicts between operating system services [61]. 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 worker instances trying to communicate with (or waiting for) another instance on the same thread.

4.3. Speeding Up a PBM Code Integrating More Mechanisms. Finally, a more complex, integrated form of the PBM code incorporating terms for consolidation, aggregation, and liquid drying/rewetting is parallelized and executed (see Appendices A, B, and C, resp., for kernels used). These mechanisms, in addition to breakage/attrition, are fundamental in describing the granulation process accurately to a greater extent. Although breakage is a crucial element, the focus is still on aggregation as it remains the most computationally intensive and therefore the primary target for parallelization in a full-fledged PBM code. 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 the aggregation terms (formation and depletion), each of which was assigned



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



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

Figure 6: Comparison of simulation times for a sequential and parallel PBM code.

to run on individual workers, thereby improving parallelism. The simulation was carried out on the INTEL Core 2 Quad Q6600, utilizing all four cores. The temporal evolution of physical properties was plotted for both the SPMD and the single CPU versions of the code (Figure 5), after which the time required for simulation of the parallel and sequential execution was plotted to display speedup (Figure 6).

As can be seen from Figure 5(a), the total number of particles predictably decreases over time due to aggregation by coalescence. The total volume of the particles, Figure 5(b), 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(c). 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. The grid size is increased and the corresponding simulation times are plotted. Even for a grid size of just 15, a speedup of 15.5 times was achieved, which is significant considering that only four workers were used. This is an example of *superlinear speedup* where for n processors, a speedup of greater than n is produced [62]. Superlinear speedup may happen if problem size per processor is small enough to fit into registers, data caches, or other smaller, yet faster memory banks instead of the RAM [63]. Since some of the parallelized functions like drying/rewetting and consolidation utilize just a few variables per processor, causes of parallel inefficiency (load imbalance, interprocessor communication) are offset resulting in faster multiplication-addition (MAD) operations than on a uniprocessor machine, where bandwidth consumption would be higher than the rate at which RAM could deliver.

5. Conclusions and Future Work

Parallel computing has been studied for several years, but its application to particulate processes described by population balance models has been limited to a few studies in crystallization [24, 25]. 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 approaches for parallelism as appropriate; and finally optimizing for minimal variable transfer overhead. We have proposed here two methods of efficiently parallelizing the integrodifferential equations comprising the aggregation term for the CPU, either using loop slicing or, alternately, a brute force, fork-join method in conjunction with MATLAB's parallel Computing toolbox. This approach to parallelism provides the modelling and research community with the necessary tools to reduce simulation times with minimal effort. The results show a speedup of 2.6 times with 8 workers over a sequential code, with potential for improving speedup by increasing problem size, using distributed data types and employing constructs like `labSend` and `labReceive` for better memory read/write patterns. The corresponding increased demand for RAM can be handled algorithmically by using distributed data types and MPI-based constructs like `labSend` and `labReceive`. For the first time, a method describing the utility of GPU computing for PBMs is demonstrated. For the GPU, we utilized the JACKET toolbox for MATLAB 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 initially did not seem encouraging 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 given the significant advances made in its architecture with each new generation. Finally, a relatively more complex code integrating several mechanisms was parallelized with the aid of the `SPMD` keyword, yielding a superlinear speedup of 15.5 times. Future work will include building better parallel algorithms with efficient task scheduling for better speedup; parallelization across processors on multiple networked machines using the MATLAB distributed computing server package for CPUs; utilizing next-generation Tesla architecture-based NVIDIA GPUs with advanced architecture to attain significant speedup considering the fact that once parallelized, PBMs are well suited for execution on massively parallel architectures. Furthermore, with increased computational power in terms of RAM and processing speed, CPU and GPU parallel computing will show increased efficiency as a result of enhanced memory requirements to store and process large amounts of data from increased number of grids and/or dimensionality of the problem. Methods developed (for CPU and GPU parallel computing) can also be easily extended to other particulate processes that are described by PBMs such as crystallization, milling, and polymerization with potential to aid in computer-aided modeling and simulation and offer economic benefit to industries that deal with such processes [64].

Appendices

A. Aggregation Kernel

For our simulation purposes, we have considered the empirical aggregation kernel proposed by Madec et al. [54]. It takes into account the various parameters such as the particle size and binder volume and can be considered to be a more appropriate empirical kernel for our multidimensional PBE:

$$\beta = \beta_0 \left(L_1^3 - L_2^3\right)\left(\left(c_1 + c_2\right)^{\alpha}\left(100 - \frac{c_1 + c_2}{2}\right)^{\delta}\right)^{\alpha}, \quad \text{(A.1)}$$

where

$$c_i = \frac{\text{volume of liquid}}{\text{volume of agglomerate}} \times 100. \quad \text{(A.2)}$$

B. 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 Verkoeijen et al. [31]. It can be given as

$$\frac{d\epsilon}{dt} = -c\left(\epsilon - \epsilon_{\min}\right),$$
$$\frac{dg}{dt} = \frac{c\left(s + l + g\right)\left(1 - \epsilon_{\min}\right)}{s} \times \left[l - \frac{\epsilon_{\min} s}{1 - \epsilon_{\min}} + g\right], \quad \text{(B.1)}$$

where the porosity ϵ is

$$\epsilon = \frac{l + g}{s + l + g}. \quad \text{(B.2)}$$

Here $\epsilon_{\min}$ is the minimum porosity of the granules and c is the compaction rate constant.

C. Drying/Rewetting

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

$$\frac{dL}{dt} = \frac{\dot{m}_{\text{spray}}\left(1 - c_{\text{binder}}\right) - \dot{m}_{\text{evap}}}{m_{\text{solid}}}, \quad \text{(C.1)}$$

where

$$m_{\text{solid}} = m_{\text{solid fraction}} + \dot{m}_{\text{spray}}\, c_{\text{binder}} \Delta t. \quad \text{(C.2)}$$

In the above equations, $\dot{m}_{\text{spray}}$ is the binder spray rate, c_{binder} is the concentration of solid binder in the slurry added, m_{evap} is the rate of liquid being evaporated (in this work $\dot{m}_{\text{evap}} = 0$, for the sake of simplicity), $m_{\text{solid fraction}}$ 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 $x_{\text{liquid}} + \delta x_{\text{liquid}}$, which cannot be 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

$$\text{fraction}\left(j\right) = \frac{X - x\left(j\right)}{x\left(j+1\right) - x\left(j\right)}, \quad \text{(C.3)}$$

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

D. Numerical Solution

Using a multidimensional population balance with an appropriate kernel ensures an improved analysis/prediction of the granulation process. But besides developing the model, incorporating an efficient numerical technique for solution to such integropartial differential equation is yet another difficult task. The multiple time scales and multiple dimensions introduce various complexities to the solution technique. Hence, it is very crucial to develop robust models with efficient solution techniques for such a framework. Our approach for obtaining a solution to such equations is based on a hierarchical two-tiered algorithm as proposed by Immanuel and Doyle III [55]. This involves using the finite volume approach for discretization with respect to each individual solid, liquid, and gas volume, followed by integration of the population balance over the domain of these subpopulations. Neglecting layering, (3), can be expressed in the discrete form as shown in

$$\frac{dF'_{i,j,k}}{dt} + \left(\frac{F'_{i,j,k}}{\Delta l_j} \frac{dl}{dt}\bigg|_{l_j} - \frac{F'_{i,j+1,k}}{\Delta l_{j+1}} \frac{dl}{dt}\bigg|_{l_{j+1}} \right) + \left(\frac{F'_{i,j,k}}{\Delta g_k} \frac{dg}{dt}\bigg|_{g_k} - \frac{F'_{i,j,k+1}}{\Delta g_{k+1}} \frac{dg}{dt}\bigg|_{g_{k+1}} \right) = \mathfrak{R}_{\text{nuc}}\left(s_i, l_j, g_k\right) + \mathfrak{R}_{\text{agg}}\left(s_i, l_j, g_k\right) + \mathfrak{R}_{\text{break}}\left(s_i, l_j, g_k\right). \quad \text{(D.1)}$$

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 ith bin along the solid volume axis, l_j is the value of the liquid volume at the upper end of the jth bin along the liquid volume axis, and g_k is the value of the gas volume at the upper end of the kth bin along the gas volume axis. Δs_i, Δl_j, and Δg_k are the sizes of the ith, jth, and kth bin with respect to the solid, liquid, and gas volume axes. Using this technique, the population balance equation is reduced to a system of ordinary differential equations in terms of the rates of nucleation ($\mathfrak{R}_{\text{nuc}}(s_i, l_j, g_k)$), aggregation ($\mathfrak{R}_{\text{agg}}(s_i, l_j, g_k)$), and breakage ($\mathfrak{R}_{\text{break}}(s_i, l_j, g_k)$). The triple integral for the aggregation term can thereby be evaluated by casting it into simpler addition and multiplication terms. Using this approach, the numerical steps are hard coded in MATLAB to obtain the final efficient solution of the population balance equation. Nucleation and breakage are included for clarity but are not considered in this study.

E. Calculation of Output Properties

Bulk properties such as average diameter, total number distribution of particles, and total volume are obtained from the simulation results in order to qualitatively and quantitatively analyse the macroscopic properties. The total number distribution of particles in the system is calculated as

$$\text{Total number distribution} = \sum_{s,l,g} F\left(s,l,g\right). \quad \text{(E.1)}$$

The total number of particles in the system is obtained by multiplying the total number distribution by Avogadro's number. Next, the total volume of particles in the system and average granule volume are used to calculate the average granule diameter, with the assumption of the particle being spherical, as

$$\text{Total volume} = \sum_{s,l,g} \left[F\left(s,l,g\right) * \left(s+l+g\right)\right],$$
$$\text{Average volume} = \frac{\sum_{s,l,g}\left[F\left(s,l,g\right) * \left(s+l+g\right)\right]}{\sum_{s,l,g} F\left(s,l,g\right)}, \quad \text{(E.2)}$$
$$\text{Average diameter} = \frac{6}{\pi} * \left(\text{Average volume}\right)^{1/3}.$$

Disclosure

The authors have any direct financial relation with the commercial identities mentioned in the paper.

Conflict of Interests

All authors state that they have no conflict of interests.

Acknowledgment

This work is supported by the National Science Foundation Engineering Research Center on Structured Organic Particulate Systems through Grant NSF-ECC 0540855.

References

[1] P. A. Cundall and O. D. L. Strack, "A discrete numerical model for granular assemblies," *Geotechnique*, vol. 29, no. 1, pp. 47–65, 1979.

[2] H. B. Matthews, S. M. Miller, and J. B. Rawlings, "Model identification for crystallization: theory and experimental verification," *Powder Technology*, vol. 88, no. 3, pp. 227–235, 1996.

[3] R. Ramachandran, C. D. Immanuel, F. Stepanek, J. D. Litster, and F. J. Doyle, "A mechanistic model for breakage in population balances of granulation: theoretical kernel development and experimental validation," *Chemical Engineering Research and Design*, vol. 87, no. 4, pp. 598–614, 2009.

[4] F. J. Muzzio, T. Shinbrot, and B. J. Glasser, "Powder technology in the pharmaceutical industry: the need to catch up fast," *Powder Technology*, vol. 124, no. 1-2, pp. 1–7, 2002.

[5] S. M. Iveson, J. D. Litster, K. Hapgood, and B. J. Ennis, "Nucleation, growth and breakage phenomena in agitated wet granulation processes: a review," *Powder Technology*, vol. 117, no. 1-2, pp. 3–39, 2001.

[6] J. A. Gantt, I. T. Cameron, J. D. Litster, and E. P. Gatzke, "Determination of coalescence kernels for high-shear granulation using DEM simulations," *Powder Technology*, vol. 170, no. 2, pp. 53–63, 2006.

[7] C. D. Immanuel and F. J. Doyle III, "Solution technique for a multi-dimensional population balance model describing granulation processes," *Powder Technology*, vol. 156, no. 2-3, pp. 213–225, 2005.

[8] J. M. H. Poon, C. D. Immanuel, F. J. Doyle, III F.J., and J. D. Litster, "A three-dimensional population balance model of granulation with a mechanistic representation of the nucleation and aggregation phenomena," *Chemical Engineering Science*, vol. 63, no. 5, pp. 1315–1329, 2008.

[9] J. M. H. Poon, R. Ramachandran, C. F. W. Sanders et al., "Experimental validation studies on a multi-dimensional and multi-scale population balance model of batch granulation," *Chemical Engineering Science*, vol. 64, no. 4, pp. 775–786, 2009.

[10] M. Dosta, S. Heinrich, and J. Werther, "Fluidized bed spray granulation: analysis of the system behaviour by means of dynamic flowsheet simulation," *Powder Technology*, vol. 204, no. 1, pp. 71–82, 2010.

[11] R. Ramachandran and P. I. Barton, "Effective parameter estimation within a multi-dimensional population balance model framework," *Chemical Engineering Science*, vol. 65, no. 16, pp. 4884–4893, 2010.

[12] R. Ramachandran and A. Chaudhury, "Model-based design and control of continuous drum granulation processes," *Chemical Engineering Research & Design*, vol. 90, no. 8, pp. 1063–1073, 2012.

[13] R. Ramachandran, M. A. Ansari, A. Chaudhury, A. Kapadia, A. V. Prakash, and F. Stepanek, "A quantitative assessment of the influence of primary particle size distribution on granule inhomogeneity," *Chemical Engineering Science*, vol. 71, pp. 104–110.

[14] J. A. Gantt and E. P. Gatzke, "A stochastic technique for multidimensional granulation modeling," *AIChE Journal*, vol. 52, no. 9, pp. 3067–3077, 2006.

[15] F. Stepanek, P. Rajniak, C. Mancinelli, R. T. Chern, and R. Ramachandran, "Distribution and accessibility of binder in wet granules," *Powder Technology*, vol. 189, no. 2, pp. 376–384, 2009.

[16] P. Rajniak, F. Stepanek, K. Dhanasekharan, R. Fan, C. Mancinelli, and R. T. Chern, "A combined experimental and computational study of wet granulation in a Wurster fluid bed granulator," *Powder Technology*, vol. 189, no. 2, pp. 190–201, 2009.

[17] B. Freireich, J. Li, J. Litster, and C. Wassgren, "Incorporating particle flow information from discrete element simulations in population balance models of mixer-coaters," *Chemical Engineering Science*, vol. 66, no. 16, pp. 3592–3604, 2011.

[18] R. Ramachandran, J. Arjunan, A. Chaudhury, and M. Ierapetritou, "Model-based control-loop performance assessment of a continuous direct compaction pharmaceutical process," *Journal of Pharmaceutical Innovation*, vol. 6, pp. 249–263, 2011.

[19] P. G. Raeth and J. C. Chaves, "Parallel matlab using standard mpi implementations," in *Proceedings of the High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC '10)*, pp. 438–441, June 2010.

[20] R. Panuganti, *A high productivity framework for parallel data intensive computing in MATLAB [Ph.D. thesis]*, The Ohio State University, 2009.

[21] R. Swinburne, *How to Overclock the Intel Core I5-2500k*, 2011.

[22] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: domain decomposition methods in hybrid CPU-GPU architectures," *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13–16, pp. 1490–1508, 2011.

[23] Z. Bai-Da, T. Yu-Hua, W. Jun-Jie, and L. Xin, "Speeding up the matlab complex networks package using graphic processors," *Chinese Physics B*, vol. 20, Article ID 098901, 2011.

[24] R. Gunawan, I. Fusman, and R. D. Braatz, "Parallel high-resolution finite volume simulation of particulate processes," *AIChE Journal*, vol. 54, no. 6, pp. 1449–1458, 2008.

[25] S. Ganesan and L. Tobiska, "An operator-splitting finite element method for the effcient parallel solution of multidimensional population balance systems," *Chemical Engineering Science*, vol. 69, no. 1, pp. 59–68, 2012.

[26] C. A. Radeke, B. J. Glasser, and J. G. Khinast, "Large-scale powder mixer simulations using massively parallel GPUarchitectures," *Chemical Engineering Science*, vol. 65, no. 24, pp. 6435–6442, 2010.

[27] P. C. Kapur, "A coalescence model for granulation," *Iandec Process Design and Development*, vol. 8, no. 1, pp. 51–56, 1969.

[28] A. D. Salman, M. J. Hounslow, and J. P. K. Seville, "Preface," *Handbook of Powder Technology*, vol. 11, pp. xi–xii, 2007.

[29] A. Annapragada and J. Neilly, "On the modelling of granulation processes: a short note," *Powder Technology*, vol. 89, no. 1, pp. 83–84, 1996.

[30] S. M. Iveson, "Limitations of one-dimensional population balance models of wet granulation processes," *Powder Technology*, vol. 124, no. 3, pp. 219–229, 2002.

[31] D. Verkoeijen, G. A. Pouw, G. M. H. Meesters, and B. Scarlett, "Population balances for particulate processes—a volume approach," *Chemical Engineering Science*, vol. 57, no. 12, pp. 2287–2303, 2002.

[32] D. Ramkrishna, *Population Balances*, Theory an Applications to Particulate Systems Engineering, Elsevier Science, 2000.

[33] R. Buyya, *The Design of Paras Microkernel*, 1998.

[34] ORACLE, "An oracle white paper: parallel programming with oracle developer tools," Tech. Rep., Oracle Corporation, Redwood Shores, Calif, USA, 2010.

[35] SPEC, *Third Quarter 2011 SPEC CPU2006 Results*, Standard Performance Evaluation Corporation, Gainesville, Fla, USA, 2011.

[36] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 1st edition, 1999.

[37] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison Wesley, 2nd edition, 2003.

[38] I. Foster, *Designing and Building Parallel Programs*, Addison Wesley, 1st edition, 1995.

[39] B. Barney, *Introduction to Parallel Computing*, 2011.

[40] W. Gao and Q. Kemao, "Parallel computing in experimental mechanics and optical measurement: a review," *Optics and Lasers in Engineering*, vol. 50, no. 4, pp. 608–617, 2011.

[41] R. Duncan, "Survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, 1990.

[42] S. Siewert, "Using intel streaming simd extensions and intel integrated performance primitives to accelerate algorithms," Tech. Rep., Atrato, 2009.

[43] MathWorks, *Parallel Computing Toolbox: Product Description*, MathWorks, Natick, Mass, USA, 2011.

[44] P. Luszczek, "Parallel programming in MATLAB," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 277–283, 2009.

[45] F. Warg, *Techniques to reduce thread-level speculation overhead [Ph.D. thesis]*, Department of Computer Science and Engineering, Chalmers University Of Technology, 2006.

[46] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Programming with the Message-Passing Interface*, Massachusetts Institute of Technology press, 2nd edition, 1999.

[47] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors: A Hands on Approach*, Morgan Kaufmann, 2010.

[48] Nvidia Corporation, *NVIDIA GeForce GTX 200 GPU Architectural Overview: Second-Generation Unified GPU Architecture for Visual Computing*, 2008.

[49] NVIDIA Corporation, *NVIDIA CUDA Programming Guide*, version 3.0 edition, 2010.

[50] H. Chafi, A. K. Sujeeth, K. J. Brown, H. J. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, pp. 35–46, ACM, New York, NY, USA, 2011.

[51] F. Bouchez, "Research report: GPGPU and Matlab," Tech. Rep., Indian Institute of science, Bangalore, India, 2010.

[52] Accelereyes. Jacket 2. 0 documentation, 2011.

[53] A. Klockner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: a scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 911, pp. 1–24, 2011.

[54] L. Madec, L. Falk, and E. Plasari, "Modelling of the agglomeration in suspension process with multidimensional kernels," *Powder Technology*, vol. 130, no. 1–3, pp. 147–153, 2003.

[55] C. D. Immanuel and F. J. Doyle III, "Computationally efficient solution of population balance models incorporating nucleation, growth and coagulation: application to emulsion polymerization," *Chemical Engineering Science*, vol. 58, no. 16, pp. 3681–3698, 2003.

[56] B. Shankar, L. Roh, W. Bhm, and W. Najjar, *Control of Loop Parallelism in Multithreaded Code*, 1995.

[57] R. Refianti, R. Refianti, and D. T. Hasta, "Workshare process of thread programming and mpimodel on multicore architecture," *International Journal of Advanced Computer Science and Applications*, vol. 2, pp. 99–107, 2011.

[58] M. D. Haines, *Distributed runtime support for task nd data management [Ph.D. thesis]*, Colorado State University, 1993.

[59] Y. Zhang, F. Mueller, X. Cui, and T. Potok, "Data-intensive document clustering on graphics processing unit (GPU) clusters," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 211–224, 2011.

[60] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," Tech. Rep., Department of ECE, Duke University, 2011.

[61] R. Brightwell, W. Camp, B. Cole et al., "Architectural specification for massively parallel computers: an experience and measurement-based approach," *Concurrency Computation Practice and Experience*, vol. 17, no. 10, pp. 1271–1316, 2005.

[62] S. G. Akl, "Superlinear performance in real-time parallel computation," *Journal of Supercomputing*, vol. 29, no. 1, pp. 89–111, 2004.

[63] J. L. Gustafson, G. R. Montry, R. E. Benner, and C. W. Gear, "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, pp. 609–638, 1988.

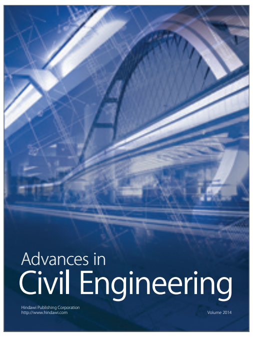
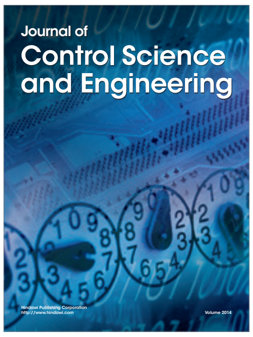
[64] S. D. Schaber, D. I. Gerogiorgis, R. Ramachandran, J. M. B. Evans, P. I. Barton, and B. L. Trout, "Economic analysis of integrated continuous and batch pharmaceutical manufacturing: a case study," *Industrial and Engineering Chemistry Research*, vol. 50, pp. 10083–10092, 2011.

Journal of
Engineering

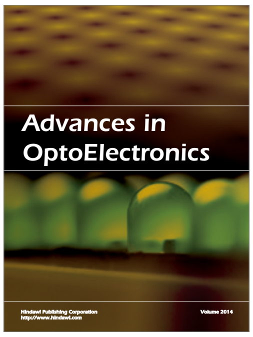
The Scientific World Journal

International Journal of
Rotating Machinery

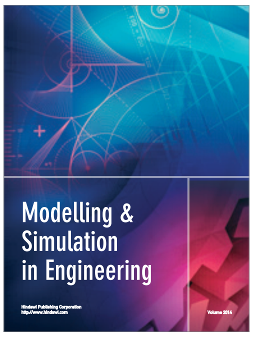
Journal of
Sensors

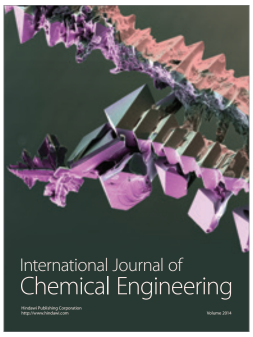
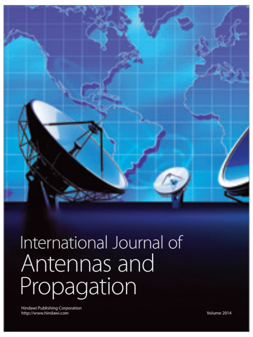
International Journal of
Distributed Sensor Networks

Advances in
Civil Engineering

Journal of
Control Science and Engineering

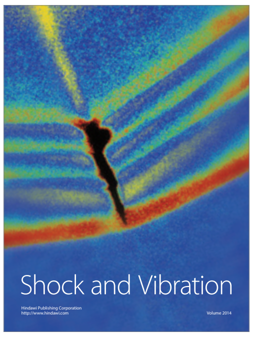
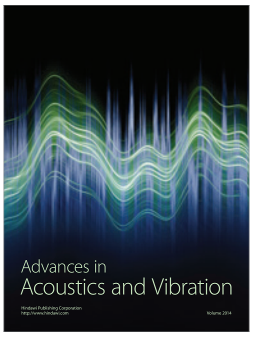
Journal of
Robotics

Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration