MODELING HETEROGENEOUS MESOSCOPIC FLUIDS IN IRREGULAR GEOMETRIES USING SHARED MEMORY SYSTEMS

Krzysztof Boryczko$^{1,2}$, Witold Dzwinel$^1$, David A. Yuen$^2$

$^1$Institute of Computer Science, AGH University of Technology, Mickiewicza 30, 30-059 Kraków, Poland

$^2$Minnesota Supercomputing Institute, University of Minnesota, MN 55455, Minneapolis, MN, USA

boryczko@uci.agh.edu.pl

Abstract

We discuss the use of current shared-memory systems for discrete-particle modeling of heterogeneous mesoscopic fluids in irregular geometries. This has been demonstrated by way of mesoscopic blood flow in bifurcating capillary vessels. The plasma is represented by fluid particles, while the other blood constituents are made of "solid" particles interacting with harmonic forces. We show that irregular boundary conditions and heterogeneity of the particle fluid inhibit efficient implementation of the model on superscalar processors. These data structures render useless many architectural issues concerning data independence, such as parallel pipelining, branch prediction, and out-of-order execution. In employing MPI on shared memory machines, we have constructed a simple middleware library to simplify parallelization. The particle code was tested on 4 and 8 processors of SGI/Origin 3800 (R14000/500), IBM Regatta (Power4/1300), SGI Altix 3000 (Itanium®2/1300) systems and two-processor AMD Opteron 240 motherboard. The tests were performed for the same system employing 2 million fluid and "solid" particles with and without load-balancing. The efficiency of the particle code depends critically on the memory latency. Therefore, the latest architectures with the fastest CPU-memory interface, such as AMD Opteron and Power4, represent the most promising platforms for modeling the complex mesoscopic systems with fluid particles. By sacrificing computational efficiency at the expense of simpler but portable solvers, we can promote the present trends in the development of easy-to-use computational services based on GRID paradigm.

Keywords: parallel algorithm, load-balancing, fluid particle model, blood flow dynamics, shared memory systems

1 Introduction

Mesoscopic modeling of the physical and chemical processes can be carried out by using the discrete-particle paradigm. Attacking this sort of problems by using non-equilibrium molecular dynamics (NEMD) approach [Boon and Yip, 1991] requires multi-billion particle ensembles and high-performance computing tools, such as distributed memory systems consisting of hundreds and thousands of processors [Vashishta et. al., 2002]. As shown in [Flekkoy and Coveney, 1999], in the mesoscale the fast modes of particle motion can be eliminated or averaged out in favor of a coarse-grain representation – fluid particles. This level of granularity allows for modeling mesoscopic phenomena by using a smaller ensemble of fluid particles instead of a large number of discrete atoms.

In [Boryczko et al., 2002] we presented the results from parallel computations of the fluid particle model (FPM) and the results of its implementation on two multiprocessor platforms of different architecture: distributed memory machine IBM SP with Power3+/375 and NUMA (non-uniform memory access) SGI/Origin 3800 with R4000/500. The speedup of 13 and 26 on 32 processors of IBM SP and SGI/Origin 3800, respectively, was reported for 16 million fluid particles in 3D. It is considerably lower than for optimized distributed memory NEMD codes [Vashishta, 2003]. The reasons are that our code was not tuned for a specific computer architecture and the memory and communication loads required per FPM particle is much greater than for MD atom. We showed that the effect of small cache dictates the program efficiency on both machines, but communication overhead is distinctly lower on a virtually shared memory SGI/Origin than on a distributed memory IBM SP.

The results discussed in [Boryczko et al., 2002] come from using the version of fluid particle code, which was originally designed for geometrically regular problems. Implementations of discrete particle methods in simulating flows in irregular geometries, such as red blood cells clotting in bifurcating capillaries, involve using load-balancing schemes. However, due to the load-balancing, the code will require greater communication bandwidth. Consequently, more efficient communication schemes between local memories of distributed memory system (see Fig.1) have to be implemented to realize message passing paradigm.

Instead of tuning the communication schemes, thus making them more and more platform dependent, we use instead shared memory systems, such as the virtual (NUMA) and global shared memory (GSM) architectures for which processes communicate using main memory (see Fig.1). Shared memory is an efficient tool of passing data between processes. One process will create a memory portion, which other processes can access. The attaching procedure must have the appropriate permissions. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation.

This issue is reasonable due to reduction of the problem size of mesoscopic modeling - partially at the expense of more complex functional model of interactions - by a coarse-graining procedure. Smaller number but powerful processors with large memory allows for simplifying sophisticated domain decomposition used in NEMD [Nakano et. al., 2001; Vashishta et.al., 2002]. Consequently, complicated communication algorithms and load-balancing procedures, which are designed for achieving a maximal efficiency for distributed memory systems, can be replaced with simpler schemes.

The paper is organized as follows. First, we introduce and discuss the shared memory platforms and processors we have employed. Second, we describe briefly the main assumptions of the parallel fluid particle algorithm. Then we present the load-balancing procedures and analyze the timings. Finally, we discuss the conclusions.
Fig.1 Distributed versus shared memory architectures: cache coherent non-uniform memory access (ccNUMA) and global shared memory – uniform memory access machines (GSM-UMA).

2 Principal features of selected shared memory systems

A slow memory access time poses a huge bottleneck for achieving high efficiency for problems, such as N-body simulations both on distributed memory and shared memory systems. There exist many CPU-memory interfaces on shared memory machines involving cache memories of various architectures and sizes, a choice of memory bandwidths, memory latencies and CPUs-memory interconnections. To find the most suitable architecture, we measure the efficiency of fluid particle code running on the following multiprocessor architectures:

1. ccNUMA (cache coherent non-uniform memory access) SGI/Origin 3800 with R14000/500 processors supported with two level cache and relatively large L2 cache (8 MB),
2. SMP(symmetric multiprocessor) IBM Regatta with Power4/1300 processors supported with three level cache with much smaller L2 (1.44 MB shared by two processors) but large 32 MB L3 cache,
3. GSM (global shared memory) SGI Altix 3000 system with Itanium®/2/1300 processor with small L2 and L3 caches (256 kB and 3 MB, respectively) but very fast access to the main memory.
4. Opteron 240 motherboard with two AMD Opteron 1.4 GHz processors and small L2 cache (1MB) but very fast access to the main memory.

The basic features of these systems are shown in Table 1.
Table.1 Basic parameters of SGI Origin 3800, IBM Regatta, SGI Altix systems and Opteron 240 motherboard

<table>
<thead>
<tr>
<th>SGI/Origin 3800</th>
<th>MIPS R14000/500 MHz</th>
<th>IBM Regatta pSeries 690 Turbo</th>
<th>Power4 1300 MHz</th>
<th>SGI Altix 3000 (Linux cluster)</th>
<th>Itanium®2 1300 MHz</th>
<th>Opteron 240 motherboard 1400 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of processors</td>
<td>128</td>
<td>1</td>
<td>48</td>
<td>1</td>
<td>48</td>
<td>1</td>
</tr>
<tr>
<td>Nodes</td>
<td>32</td>
<td>-</td>
<td>4 (MCMs)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Memory per node</td>
<td>8 GB (4 proc.)</td>
<td>-</td>
<td>8 GB (8 proc.)</td>
<td>-</td>
<td>2 GB (1 proc.)</td>
<td>-</td>
</tr>
<tr>
<td>No. on-chip proc.</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>2 (cores)</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Architecture</td>
<td>ccNUMA</td>
<td>RISC 64</td>
<td>SMP</td>
<td>RISC 64</td>
<td>GSM (Global Shared Memory Cluster)</td>
<td>EPIC</td>
</tr>
<tr>
<td>Cache on chip</td>
<td>-</td>
<td>L1 (32/32kB) L2 (8MB)</td>
<td>-</td>
<td>L1 (32/64kB) L2 (1.44MB) L3 (32 MB) L3 is off-chip L2, L3 shared by 2 cores</td>
<td>-</td>
<td>L1(32/32) L2(256kB) L3(3MB)</td>
</tr>
<tr>
<td>CPU-Memory bandwidth</td>
<td>3.2 GB/s</td>
<td>13.86 GB/s (L3-Main Memory) shared by 2 cores</td>
<td>6.4 GB/s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory latency</td>
<td>180 ns (close) 290-440ns (distant) closest memory shared by only 4 processors</td>
<td>178 ns processor-memory communication bandwidth shared by 32 processors</td>
<td>50 ns SGI NUMAlink interconnect</td>
<td>65 ns Inter-processor latency 140 ns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table.2 CPU speed and throughput on SGI Origin 3800, IBM Regatta, SGI Altix systems and Opteron 240 motherboard [http://www.specbench.org/cpu2000/results/]

<table>
<thead>
<tr>
<th>SGI/Origin 3800 (throughput)</th>
<th>MIPS R14000/500 MHz (speed)</th>
<th>IBM Regatta pSeries 690 Turbo (throughput)</th>
<th>Power4 1300 MHz (speed)</th>
<th>SGI Altix 3000 (throughput)</th>
<th>Itanium®2 1300 MHz (speed)</th>
<th>Opteron 240 motherboard 1400 MHz (speed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of processors</td>
<td>128</td>
<td>1</td>
<td>32</td>
<td>1</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>PeakPerformance (RMax)</td>
<td>128 GFlop/s</td>
<td>1 GFlop/s</td>
<td>83.2 GFlop/s</td>
<td>2.6 GFlop/s per core (5.2 GFlop/s per chip)</td>
<td>166.4 GFlop/s</td>
<td>5.2 GFlop/s (per 1 processor)</td>
</tr>
<tr>
<td>LINPACK n=100</td>
<td>n/a</td>
<td>427 Mflop/s</td>
<td>n/a</td>
<td>1070 Mflop/s (average)</td>
<td>n/a</td>
<td>3800 Mflop/s (best, RMax)</td>
</tr>
<tr>
<td>SPEC CFP2000</td>
<td>532-570</td>
<td>436-463</td>
<td>251-260</td>
<td>1202-1266</td>
<td>541</td>
<td>1770-1780</td>
</tr>
<tr>
<td>SPECCINT2000</td>
<td>582-605</td>
<td>410-427</td>
<td>232-249</td>
<td>804-839</td>
<td>311</td>
<td>875</td>
</tr>
</tbody>
</table>

The IBM Regatta [http://www.utexas.edu/research/] machine has a typical SMP architecture with uniform access to operational memory (UMA machine). The basic building block of this pSeries 690 Turbo system is a single Power4 chip. Each chip has two cores, or CPUs, that share a common L2 cache. The path to memory through the L3 cache (13.8 GB/s) is also shared by both cores. Each core processor has 2 floating point units each of which perform a "fused" multiply-add operation per clock cycle. At a clock speed of 1.3GHz, four floating point operations per cycle can deliver 5.2GFlop/s from each processor. The next architectural level for Power4 systems is the Multi-Chip Module (MCM). Connecting 4 dies on a multichip module forms an
"HPC" "Turbo" 8-way (dual core) SMP that shares all L3 and memory modules associated with
the MCM. Chip-to-chip interconnect "controller", interconnects 4 chips to form a distributed
crossbar switch. These interconnect fabric controllers allow each Power4 chip to access the L3
cache and memory of the other 3 chips. There exists also the MCM-to-MCM interconnection that
allows nodes to be built-up with multiple MCM modules (up to 4). However, in our test we use
only a single 8-processor MCM board.

SGI Altix 3000 servers [http://www.sgi.com/servers/altix/][1] and superclusters featuring the
new Intel Itanium®2 processors (1.30 GHz with 3M L3 cache) are Linux-based machines. The
Itanium represent the new IA64 technology and EPIC (explicitly parallel instruction computer)
architecture. This type of architecture is characterized by large instruction words of 128 bits that
contain 3 41-bit instructions and a 5-bit template that aids in steering and decoding the
instructions. The two load/store units fetch two instruction words per cycle so six instructions per
cycle are dispatched. The Itanium has also in common with these systems in that the scheduling
of instructions, unlike in RISC processors, is not carried out dynamically at run time but rather by
the compiler. Instead of large cache memories, the L2cache-memory latency is very low for
Itanium (see Table 1). Moreover, the SGI Altix 3000 family of servers and superclusters brings a
new scalability feature to Linux clusters - global shared memory. Systems with global shared
memory allow for accessing all data in the system's memory directly and efficiently, without
having to move data through I/O or networking bottlenecks. Global shared memory requires a
sophisticated system memory interconnect, like the SGI NUMAlink and application libraries, that
enable shared-memory calls, such as MPT and XPMEM from SGI.

The AMD64 architecture [http://www.tomshardware.com/cpu/20030422/opteron-06.html][2]
extends the 32-bit register of the IA-32 processors to 64-bit Opteron processor (clock speed
ranging from 1.4-2 GHz). Instead of the usual parallel FSB, the CPU communicates via a
HyperTransport interface. The serial interface with a variable bit-rate allows the SledgeHammer
to attain a data transfer rate of 3.2 GB/s - in both directions simultaneously. This results in a total
bandwidth of 6.4 GB/s per Hypertransport port. The entire data traffic of the Opteron processor
runs through the HyperTransport interface and the integrated memory controller. In order to let
the neighboring CPU gain direct access to its system memory, the Opteron uses the XBAR
switch. For commands and addresses, the XBAR switch has additional 64-bit buses available.
The memory latency achieved by the Opteron is 65ns. The inter-processor latency is under 140ns.

The SGI/Origin 3800 [http://www.nsc.liu.se/sgi3k/recent.html][3] is an older system
constructed on the basis of MIPS R14000/500 processors. They are interconnected into 4
processors SMP nodes, which form 3-D cubes supernodes. SGI/Origin 3800 is ccNUMA (cache
coherent non uniform memory access) machine with virtual shared memory and with highly
optimized distant memory calls. They must be optimized due to the calls to the slow, distant
memory (located on distant nodes) are the main source of the overheads on ccNUMA systems.
The latency of L2cache-memory is 180ns for a local memory access and 290-440ns for distant
calls. It is longer than for Regatta, which is only 178 ns, however, communication bandwidth on
32-way node must be shared between 32 processors instead of 4 on an internal node of
SGI/Origin machine. Basic parameters for the four systems are presented in Table1.

The sustained performance of SPEC CPU2000 per single processor for the IBM Regatta
system is 1.8 (for multitask tests – throughput) to 2.8 (for single processor and single task tests –
speed) faster than for a single R14000/500 of SGI/Origin 3800 [www.specbench.org]. The first
number means that for intensive communication and multitasking a 32-way turbo node (2
processors on chip) has too a slow memory bandwidth, especially for benchmarks with highly
localized data structures, for which NUMA architecture is better suited. However, for a 16-way
IBM pSeries 690 HPC (16 processors, 1 processor on chip) with greater memory bandwidth the throughput is higher and is 2.6 times greater per processor performance than the throughput for SGI/Origin. We note that this ratio is smaller than the ratio between the frequencies of the two processors (2.8). Despite the same frequency, due to more efficient architectural issues (fast shared global memory) SGI Altix is almost 2 times faster than IBM Regatta (in throughput) while a single Itanium®2 is 50% faster than Power4 (in speed). The benchmarks from Table 2 show that the AMD Opteron is slower both than Power4 and Itanium®2. However, having 3 times shorter memory latency than the Power4 and almost the same as Itanium®2 but 4 times larger cache, the Opteron processor can beat both processors for problems with irregular geometry, which require a high memory load.

3 Particle model of blood flow in capillary vessels

The equations of macroscopic fluid dynamics for blood [Berger, 1992; Maury, 2002; Quarteroni et. al., 2002] describe the motion and energetics of homogeneous fluid over a lengthscale over 1 mm. Conversely, mesoscopic flows of heterogeneous fluid are modeled in spatial scales of 10-200µm [Dzwine et. al., 2002, 2003; Boryczko et. al., 2003]

As we have shown in [Dzwine et. al., 2003, Boryczko et., al, 2003], the mesoscopic blood system can be composed of plasma represented by fluid particles carrying other blood constituents. The capillary walls and blood components, i.e., fibrins and red blood cells, are made of “solid” particles. In Fig.2 we show the particle system representing capillary wall.

We consider here a three-dimensional system, which consists of M both fluid and “solid” particles. The particles are represented by their centers of mass, which posses several attributes, as mass, position, translational and angular velocities and type. The particles of various types are distributed within the simulation domain according to a given scenario. Particles interact with each other by different forces depending on their physical characteristics.

Fluid particles interact with forces consisting of a sum of conservative force \( F^C \), two dissipative components \( F^T \) and \( F^R \) and the Brownian force \( \tilde{F} \) [Espanol, 1998], that is

\[
F_{ij} = F_{ij}^C + F_{ij}^T + F_{ij}^R + \tilde{F}_{ij}
\]

The conservative and dissipative forces are the functions of distance between particles \( r_{ij} \) and their relative translational and angular velocities. The Brownian force is of a stochastic nature, which is defined by the symmetric, antisymmetric and trace diagonal random matrices with independent Wiener increments [Espanol, 1998]. The forces are short-ranged, i.e., there exists a cut-off radius \( r_{cut} \), which defines the range of FPM particle interactions. For \( r_{ij}>r_{cut} \), \( F_{ij}=0 \).

Fluid particles interact with “solid” particles representing wall, red blood cells and fibrins with DPD (dissipative particle dynamics) force [Hoogerbrugge and Koelman 1992]. The “solid” particles making up the various system components such as, capillary walls, red blood cell and fibrin, interact with neighboring particles of the same type with harmonic forces (see for details [Boryczko et.al., 2003]) to mimic elastic properties of the wall and red blood cells. The particles from isolated constituents (e.g., wall and red blood cell particles, particles from different red blood cells) interact with each other via FPM force with a strongly repulsive conservative component. In Fig.3 we display the heterogeneous particle system, which produces a clot of red blood cells entangled by fibrinogen, one of the most important ingredients in blood clotting. Therapies to remove clots both through enzymatic and mechanical approaches require
consideration of the biochemistry and structure of blood clots in conjunction with local transport phenomena [Diamond, 1999].

As shown in [Dzwinel et. al., 2003, Boryczko, at. al., 2003], accurate 3D simulations of blood flowing in capillary vessels require a large ensemble of particles exceeding two million. For calculating interparticle forces we have used standard approach described in [Hockney and Eastwood 1981]. As shown in Fig.4, the box is divided onto cubic cells of the edge size equal to cut-off radius. The force on a given particle includes contribution from all the particles that are closer than the value of cut-off radius and which are located within the cell containing the given particle or within the adjacent cell. Then the forces are computed by using an $O(M)$ order link-list scheme [Hockney and Eastwood, 1981].

![Fig.2 Bifurcating capillary vessels. The particles represent endothelial cells. They interact one with another via harmonic force to mimic the physics of elastic properties of the walls.](image1)

![Fig.3 This snapshot shows the formation of a clot consisting of the red blood cells (red plates) and fibrins close to the capillary neck placed on the right hand side of the figure. The fibrinogen, which is responsible for clot formation concentrates mainly in the stagnation points of the flow, producing fibrins entangling with the red blood cells. The fluid and wall particles are not shown in this figure.](image2)

The temporal evolution of the entire particle system is described by the Newtonian equations of motion. Because of the random Brownian forces, the equations of motion are now stochastic
differential equations (SDE), which are computationally more difficult than ordinary differential equations. Numerical integration of SDE by using classical Verlet scheme [Hockney and Eastwood, 1981] generates large numerical errors and artifacts, e.g., resulting in unacceptable temperature drift with simulation time [Vattulinen et al., 2002]. Therefore, very tiny timesteps must be used to obtain a reasonable approximation to the thermodynamical quantities. In contrast to dissipative particle dynamics (DPD) [Hoogerbrugge and Koelman 1992] in fluid-particle model the angular velocities and torques for each fluid particle have to be computed. More complex functional dependences of forces formula produce additional overhead. We note that the forces and torques in FPM depend not only on the particle positions and its translational velocity but also on the angular velocities. The predictor-corrector numerical schemes must be used twice per timestep.

![Fig.4 The linked-cell algorithm in 2D. In 3D the number of neighboring cells is greater but the idea is the same.](image)

In parallelizing the code we have employed the domain-decomposition procedure [Fox, 1987; Foster, 1995]. The total volume of the computational periodic box is divided into P overlapping subsystems of equal volume (Fig.5). Each subsystem is assigned to a single processor in a P processors array. The processors follow an identical predetermined sequence of operations in calculating the forces associated with the particles within assigned domain. From cells, which are situated on the boundaries between processor domains, we copy all the particle attributes to the neighboring processor. The number of particles located in the boundary cells defines the communication overhead. Slicing the box along the z-axis simplifies greatly the routing of messages. It allows also sending them in unimpeded fashion. Each processor sends the message only along one direction to its closest neighbor. This domain decomposition algorithm also matches the data parallel programming paradigm used on shared memory systems. The memory blocks containing the arrays and other data structures from neighboring domains can be also placed close to each other in the main memory address space. The memory cells with the physical attributes of the particles located close to the domain borders, can be defined as being shared with read and write permissions.

In the case of the red blood cells flowing in a capillary (as shown in Figs.3,5), equal box sections will contain different number of particles. Thus the slowest processor dealing with the largest number of particles will dictate the code efficiency. As shown in Fig.5b, the load-balancing in this case is accomplished by shifting the boundaries of processor domains, while
keeping the same number of particles in each section of the computational box [Boryczko et. al., 1994; Hayashi, at. al., 2000; Deng et. al., 2000]. From the point of data-parallel shared memory programming, this will correspond to changing write and read permissions for particles moving from one domain to the other.

**Fig.5** a) The schematics of the geometrical decomposition of the computational box. b,c) The decomposition of the computational boxes for red blood cells flow in a curved capillary without (b) and with (a) load-balancing.

We show the flowchart of the load-balancing in Fig.6. The cell size (see Fig.4) defines the smallest spatial step in which one can shift the boundaries between the processor domains. The CPU times \( t_{li} \) from each process \( l \) (where \( l=1,\ldots,P \)) are collected during subsequent timesteps of simulation and averages \( t_l \) are computed. The differences \( \Delta t_l = t - t_l \) where \( t = 1/P \sum_{i=1,P} t_i \) govern the degree of load imbalance. Because of this problem, we must set up the new borders between the processor domains.

Discrete particle model code was written in Fortran95. Our programs run on shared memory platforms and the small number of required processors. The efficient use of SPMD (single program multiple data) data-parallel programming paradigm with Pthreads, OpenMP or high performance Fortran (HPF) [Briguglio et. al., 2001] for particle codes is not trivial. For unbalanced systems it causes poor performance [Boryczko et. al., 1996, Thornley, 1998; Thornley et. al., 1999]. The problems with thread scheduling appear for many nested branch instructions. Moreover, using message passing interface (MPI) library gives a better portability to the code. However, we understand that when designing a message passing algorithm, the writer assumes a distributed memory model. Replication of data is necessary. Genuine shared memory programming is totally different. It makes very different assumptions on the breakdown and distribution of data. Moreover, MPI library involves using too many subroutines with many parameters. Therefore, we have created an easy-to-use middleware library. As shown in Fig.7, it uses the MPI procedures but simplifies the implementation of interprocessor communication and allows for instantaneous monitoring of parallel computations.
TIME AVERAGES
Compute averages $t_i$ of CPU time on each processor $i$ every LB timestep

IF $i > LB$

COMPUTE TIME DIFFERENCES
Compute mean value $T$ from the averages $t_i$, and differences $\Delta t_i$ between $T$ and all $t_i$

IF max$|\Delta t_i| >$ Threshold

FIND A NEW DECOMPOSITION OF THE COMPUTATIONAL BOX
$x_n$ shifted correspondingly to $\Delta t_i$
$\tilde{x}_n$ - new border between $n$ and $n+1$ domains

---

**Fig. 6** The flowchart of the load-balancing scheme. The mean execution CPU times $t_i$ in a given number of timesteps LB are calculated for processes corresponding to the computational domains delineated by $x_1$-$x_3$ boundary. The differences between average execution times for all the processes $t$ and the mean times for each process $t_i$ are used to set up new $\tilde{x}_1$ - $\tilde{x}_3$ boundaries between the domains.

---

**Fig. 7** The communication interface library devised as an interface between original MPI library and the code. It simplifies greatly the calling of the message-passing procedures and creates a trace file of .log type for monitoring the distributed processes.
4 Timings and tests on shared memory systems

We have performed 4 and 8 processor simulations on the four systems of the same model of blood flow along bifurcating capillary vessel for $M=2\times10^6$ particles (see Fig. 5b,c). We have activated an aggressive –O3 option for the compilers. Examples of the timings from these simulations are displayed in Figs.8-10. The average sphere of a given cut-off radius includes 15 fluid particles.

In Figs.8-10 we present the timings for the code running on 4 and 8 processors of the SGI/Origin 3800, IBM Regatta and SGI Altiix 3000 computers. The plots from the left hand side deal with the case in which a computational box was decomposed uniformly on each processor (see Fig.5a). We demonstrate that, the CPU time needed for completion the tasks on separate processors can differ even four times. The process of the longest execution time decides about the execution time of the entire code. To eliminate this deleterious effect, a load-balancing scheme must be implemented.

As shown in the plots on the right hand side of Figs.8-10 the efficiency of the code with active load-balancing procedure increases twice over the average. Comparing the timings for various frequencies of invoking load-balancing scheme (for every 50th to 200th timestep) we cannot observe any difference on SGI Origin. This means there is a small communication overhead resulting from the load-balancing. It is somewhat greater for IBM Regatta than for SGI/Origin 3800, due to faster memory access to the data residing on the local memory of the
NUMA system. We can observe large fluctuations of the code running on 4 processors of SGI Altix system (Fig.10). Because they are not detected on 8 processors, we can conclude that this may be the effect of small cache of Itanium®2 (only 256 kB). For irregular data structure involved by particle codes frequent cache misses drastically reduce its performance comparing to Power4 and AMD Opteron processors. When the processor speeds are very different and depend non-linearly on the memory load, load-balancing becomes very difficult.

The execution time for sequential FPM code on SGI/Origin 3800 (L2 8MB cache size) increases substantially after about 60 timesteps (Fig.11). This happens when the particles cross the cells boundaries and the attributes (position, velocities, forces and torques) of particles residing in the same cells (see Fig.4) are located in distant memory addresses. This overhead is small for IBM Regatta (see Fig.11), which is equipped with L3 32 MB cache (shared by 2 cores). It is completely invisible for AMD Opteron with adequately large secondary cache size and very fast access to the main memory (see Table 1). Therefore, we have found that the Opteron reaches the fastest speed for a sequential FPM code. It is almost 2 times faster than one core of the Power4 processor and Itanium®2.

The neighboring particles should also be closer one another in the computer memory. Therefore, to avoid frequent cache misses the particles are renumbered every some period of time. Consequently, the particles in the same cell are classified with consecutive numbers. However, the gap between particle numbers still exists for the particles from neighboring cells used for computing the forces. This is due to the incoherence between the linear addressing of memory and indexing of 3D particle system. Increasing the number of processors for the same number of particles, we have narrowed the gap between memory addresses of neighboring cells. However,
we can observe the super-linear speed-up resulting from a more efficient use of cache memory for a decreasing number of particles per processor. The maximum speed-ups obtained by increasing the number of processors from 4 to 8 (see Fig.5) are 2.72, 2.42 and 2.27 for SGI/Origin 3800, IBM Regatta and SGI Altix, respectively.

![Graphs showing execution time vs. time step for different processors and conditions.](image)

**Fig.10** The timings for SGI Altix 3000 machine with and without load-balancing.

In Figs.8-12 we can see the positive feedback of large and shared L3 cache of Power4 IBM Regatta system and short memory latency and relatively large secondary cache of AMD Opteron on diminishing the effect of frequent cache misses. In sum, a single Power4 processor of IBM Regatta is 2.6 times faster in average than R14000/500 SGI/Origin processor. This result is in a good agreement with SPEC benchmarks reported in Table 2. The SGI Altix is nearly two times faster than the IBM Regatta in SPEC throughput and 40% in SPEC processor speed (Table 2). Nevertheless, we find that its measured performance for our discrete particle model is even worse off than for the IBM. Comparing the timings measured on four Itanium®2 processors (Fig.10) and two-processor Opteron 240 motherboard (Fig.12), the SGI Altix is also slower per processor than AMD Opteron, which ranks the slowest in SPECFPM benchmarks from the three (see Table 2). As shown in Fig.11, the AMD Opteron is faster than the Power4 processor by a factor of two on a sequential code. However, for more processors, the IBM Regatta system uses more efficiently its large cache and appears to be the fastest.

In our opinion, discrete particles codes with irregular data structures and indirect addressing require processors with the shortest CPU-memory access, which could be reached by using large multilevel caches and/or short main memory latencies. Considering the timings of
AMD Opteron from Figs. 11, 12 and its low price/performance, we find more convincing this second issue. However, the cache size cannot be too small or too slow, as it is in Itanium® case. Many types of interactions involving many if statements almost inhibit the efficient use of powerful features of superscalar EPIC processors, such as VLIW (very long instruction word).

![Graph of Execution time for sequential FPM code](image)

**Fig. 11** Execution time for sequential FPM code (about $2.5 \times 10^5$ fluid particles) running on SGI/Origin 3800 and IBM Regatta processors.

![Graph of Timings for two-processor Opteron 240 motherboard](image)

**Fig. 12** The timings for a two-processor Opteron 240 motherboard with and without load-balancing.

4 Concluding Remarks

The GRID computing paradigm (see e.g., [Foster and Kesselman, 1998; Berman et al., 2002]) must have the following ingredients:

1. various types of specialized services situated on distributed hardware resources,
2. middleware platforms used to facilitate standardization of data formats and interoperability of data structures and programming languages.

Modeling tools and solvers must match these trends by keeping apace with the ongoing progress in the GRID environment.
Highly optimized parallel codes, running on 100 or more processors, rarely meet the common needs of the scientific community because of lack of availability and the queuing systems. They are used for benchmarking high-performance systems or they represent special purpose solvers boosting up high-performance computing. This is not a general rule and many high-performance computing centers make their resources accessible through GRID infrastructure. However, these services always will have very restricted access, and be very expensive and difficult to adapt for most client specified problems. Therefore, we suppose that the core of GRID computational services will be based on large amount of more universal solvers run on middle-ranged shared memory SMP machines and supported with user friendly interface and visualization facilities [Garbow et. al, 2003].

The problems involving multi-million particle ensembles, found in modeling mesoscopic phenomena, were considered only recently as the typical problems. Rapid increase of computational power of modern processors and growing popularity of coarse-grained discrete particle methods, such as dissipative particle dynamics, fluid particle model, smoothed particle hydrodynamics and lattice Boltzmann gas, allow for the modeling of complex problems by using smaller shared-memory systems.

The main technical problem connected with efficient use of shared memory machines in modeling of large ensembles of discrete-particles lies in highly irregular data structure they require. Many particle types, interaction forces and irregular boundary conditions involve using nesting if statements and/or indirect addressing. This strongly inhibits an efficient use of new architectural issues of modern processors, which exploit explicit parallelism involving regularity of data structures and data independence. We show that the Power4 processor with off-chip and a slow but large L3 cache can be more efficient than Itanium®2 with a modern EPIC architecture with fast access to the main memory. Moreover, the AMD Opteron with 65 ns memory latency and 1 MB L2 cache size can be 2 times faster than Power4 with >178 memory latency and 16 MB L3 cache and more than 2 times faster than Itanium®2 with 50 ns memory latency and slow 3MB L3 cache. We conclude that an effective way for decreasing high memory latency is to use the memories of the shortest latencies, reinforced with a sufficiently large secondary cache.

We have also demonstrated that, by employing simple parallel algorithms and communication schemes, we can easily adapt the discrete particle code for modeling irregular and heterogeneous problems, such as blood clotting in microscopic vessels of various shapes. We can run the same code without any changes on various shared memory servers. Flexibility, portability and efficiency of the particle model can be important in achieving of various GRID services, which will speed up mesoscopic modeling.

**Acknowledgements**

Support for this work has come from the Polish Committee for Scientific Research (KBN) project 4T11F02022 and the Complex Fluids Program of DOE.

**References**


