Graphical Visualization on Computational Simulation Using Shared Memory

Graphical Visualization on Computational Simulation Using Shared Memory

A. B. Lima[1] and Eberth Correa[2]

[1]Instituto de Ciências Tecnológicas e Exatas, Universidade Federal do Triângulo Mineiro - UFTM, 38064-200, Uberaba-MG, Brazil.
[2]Faculdade UnB Gama, Universidade de Brasília - UnB, 72444-240, Gama-DF, Brazil.

Abstract. The Shared Memory technique is a powerful tool for parallelizing computer codes. In particular it can be used to visualize the results "on the fly" without stop running the simulation. In this presentation we discuss and show how to use the technique conjugated with a visualization code using openGL.

1. Introduction

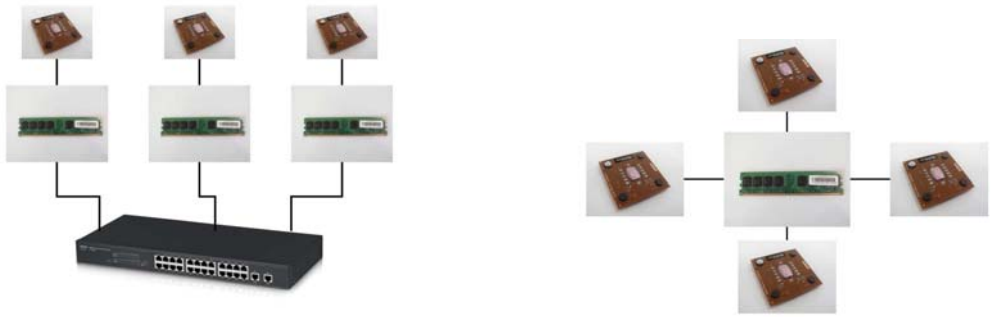
The visualization is an interesting tool to help us to understand some aspects of simulations [1–4]. In many cases we want to see the result of the simulations on the fly, so that we can decide whether the written code is correct or whether the set of chosen parameters was adequate in the actual simulation. In practice, visualization on the fly requires more computational effort than the simulation itself. As an alternative, one can dedicate some computational effort for the visualization in a separated program in such way that we can turn on or turn off the visualization when we want to. This technique is considerably advantageous, for the processing time is spent only at the visualization process.

The parallelization of the code is the more efficient way to proceed in this case. There are many ways to parallelize the code, we want to emphasize two of them: Message Passing Interface (MPI) and Shared Memory [5–8].

In the MPI technique, a copy of the code is put at each memory of each cpu and messages are exchanged between cpus (using a network connection) to set what each cpu must to do in the code. At the beginning, MPI and its library implementations (OpenMPI, MPICH, etc.) were the most useful way to implement parallelization in computer programs.

In the Shared Memory, different parts of the program use the same memory area to exchange data. Although unlike in concept, the Shared Memory parallelization has become useful with the recent development of more powerful multi-core processors found in either current personal computers or workstations, in which some of them can have up to 64 cores. It is worth emphasizing that the algorithms and programming implementation using Shared Memory is easier than MPI. A schematic view of MPI and Shared Memory is displayed in Fig. (1).

Shared Memory is one of the simplest method of interprocess communication (IPC) and allows two or more processes to exchange data accessing the same area in memory. In addition, these communications use the bus of the computer chipset being the fastest way to parallelize tasks and to avoid copying data unnecessarily.



(a) (b)

Figure 1. The (a)MPI and (b)Shared Memory schemes.The data exchange in Shared Memory is fastest that MPI.

As an example we present in the next section a standard algorithm to implement the Shared Memory in a computational simulation using Molecular Dynamics. In what follows, all the applications displayed use graphical visualizations in *openGL* for a molecular dynamics simulation of a magnetic liquid.

2. Shared Memory

To use the Shared Memory we need to allocate a memory segment. The function in C or C++ to allocate memory is called shmget ("SHared Memory GET"). Its first parameter is an integer that identifies which segment must be created. All processes can access the same memory segment by specifying this key. The second parameter specifies the number of bytes in memory segment. The third parameter is related to the flag values that specify several options to the shmget function. In the example below we show a C code fragment that creates a memory segment.

```
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHMSZ 100 //size of memory segment (100 bytes)

int main(int argc, char* argv[])
{

int key,shmid;

key  = 11; //identifies the memory segment

// shmget: the flags IPC_CREAT | 0666 check if the segment
//exist and set the segment as a read and write. See
// more flags in shmget help.
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666 )) < 0) {
                perror(''shmget'');
                return 1;
        }
```

```
return 0;
}
```

Next we need to make the Shared Memory segment available. We must use shmat,("SHared Memory ATach"). This function uses as a first argument the identifier shmid returned by the shmget.The second argument is a pointer that specifies where inside our processes address we want to map the Shared Memory. It is easier to let the operational system decide what to do. This can be done only specifying it as NULL. The third argument is a flag. We show in the following the code to be used to attach the memory segment.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int main(int argc, char* argv[])
{

double *shm;


if ((shm=(double *)shmat(shmid, NULL, 0)) == (double *) -1){
                perror(''shmat'');
                return 1;
return 0;
}
```

Now you can use the memory segment shm like a double array.

```
int main(int argc, char* argv[])
{

...

for(int i=0; i<number_of_double;i++){
shm[i]=0.0;
}

...

return 0;
}
```

To detach the Shared Memory segment we use the function shmdt.

```
int main(int argc, char* argv[])
{

...

shmdt(shm);
...
```

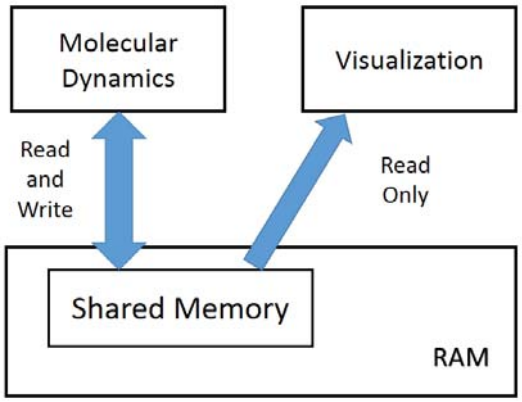


Figure 2. The Molecular Dynamics program can read and write in Shared Memory, but Visualization program just can read.

```
return 0;
}
```

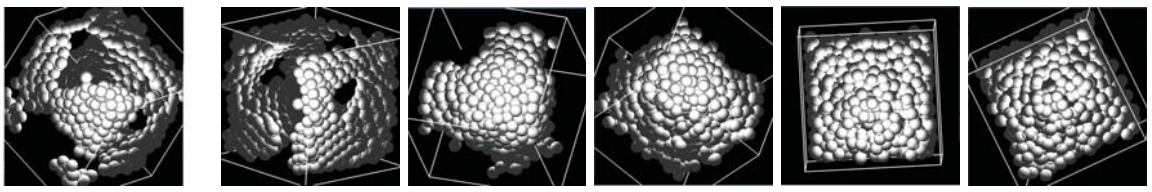
To remove a Shared Memory segment we need to use the function shmctl ("SHared Memory Control"). This function uses the id of a segment as a first parameter, a flag IPC_RMID as a second argument and a NULL pointer as a third.

```
int main(int argc, char* argv[])
{

...

shmctl(shmid,IPC_RMID,NULL);
...

return 0;
}
```

For pedagogical purposes, we employ this technique to visualize a Molecular Dynamics simulation (MD). To make a real time simulation we need to put all these functions with "create", "attach", "detach" and "destroy" Shared Memory segments into the simulation code. Typically we need to use these functions to create 3 arrays of double precision to store x, y and z, the spatial coordinates, and 3 arrays of double precision to store Vx, Vy and Vz for the velocities. Fig. (2) shows how the MD simulation and Visualization works with a Shared Memory.

In the visualization program we use the same code as before, but in the third parameter of shmget we use a flag "read only" to prevent the visualization program to change any value in the arrays of the positions or the velocities.

```
#include <sys/ipc.h>
```



(a) (b) (c) (d) (e) (f)

Figure 3. (Color online) The visualization configurations for several temperatures, e.g, $T = 0.01, 0.2, 0.4, 0.6, 0.8, 1.0$.

```
#include <sys/shm.h>

#define SHMSZ //size of memory segment

int main(int argc, char* argv[])
{

int key,shmid;
double *shm;

key  = 11; //identifies the memory segment

// shmget: the flags IPC_CREAT | 0444 check if the segment
//exist and set the segment as a read only. See
// more flags in shmget help.
if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0444 )) < 0) {
                perror(``shmget'');
                return 1;
        }

//attach the segment shm in program
if ((shm=(double *)shmat(shmid, NULL, 0)) == (double *) -1){
                perror(``shmat'');
                return 1;
//here we can read the value of shm
for(int i=0; i<number_of_double;i++){
printf(``%d/n'',shm[i]);
}

...

return 0;
}
```

To visualize the MD simulation we use the *freeglut* library in *openGL* [10]. In Fig. (3) we display some configurations after the equilibration of the system for temperatures varying from $T = 0.01$ to $T = 1.0$. Here we use reduced units [9] with the temperature given in units of

$\varepsilon/k_B \approx 120\,K$ and distances in units of σ, where ε and σ are the Lennard-Jones parameters and k_B the Boltzmann constant.

In a system with a considerable number of particles it is important to optimize the computational effort during the calculations. In a multi-core architecture the operational system does the tough task of separating the jobs between the cores. This does not forbid us to use other parallel schemes like the multithread technique, but this is out of scope of this work. With the help of the Shared Memory scheme we are able to switch on all the visualizations we want to see, separated from the computation of the MD simulations. As mentioned before we need only to enumerate the Shared Memory for each calculation in both the MD simulation and the *openGL* visualization programs without any interference between each other.

To illustrate the application we plot some radial distribution functions for the visualised configurations in Fig (4). The formation of the clusters can be seen as the temperature is decreased. However, in Fig. (3) we can see not only the clusters but also the structure formed in real time. The advantage of visualizing the simulations becomes more apparent in systems with magnetic properties like magnetic fluids [9]. Some characteristics are intrinsic for these systems like domain walls and vortex patterns in which the visualization is better suited.

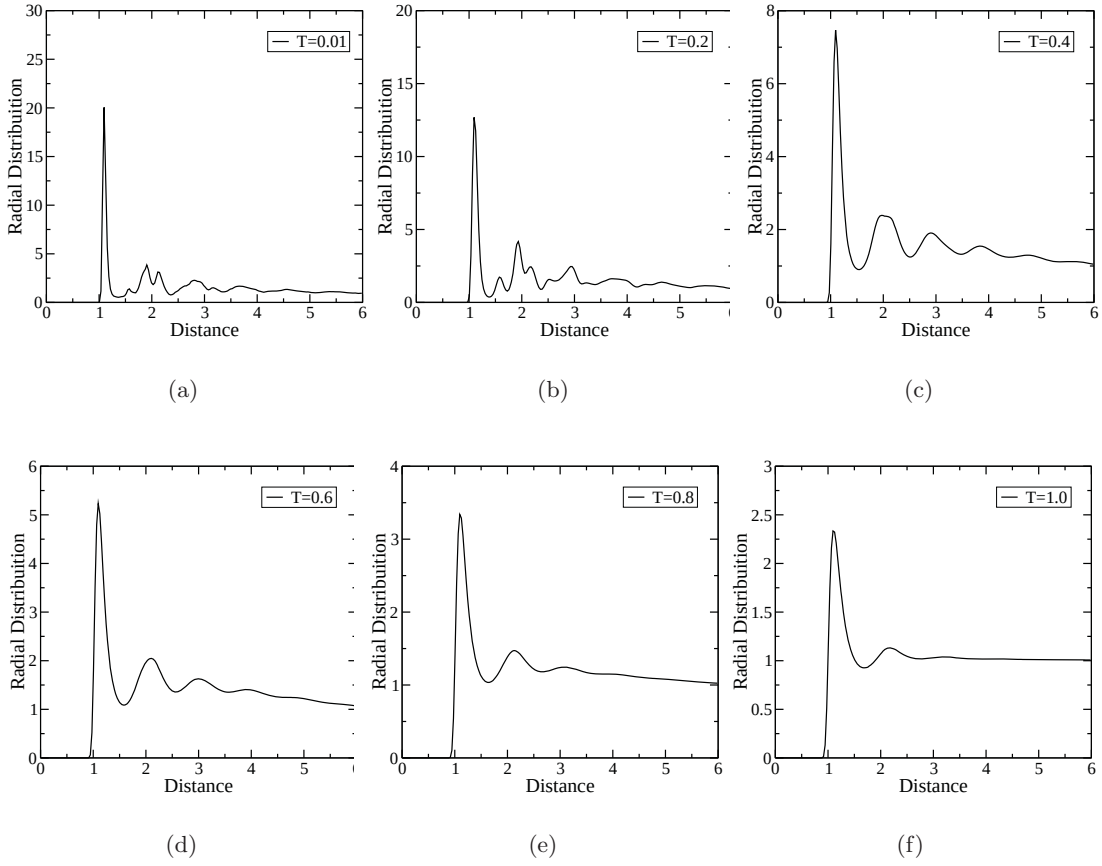


Figure 4. (Color online) The respective radial distribution functions for the configurations in Fig. 3.

3. Conclusions

The Shared Memory technique for the visualization of the simulations "on the fly" was presented. This technique for parallelization of the computation and the visualization of the simulations became accessible with the development of the multi-core technology as well as the expansion of the *RAM* memory. With the help of the *openGL* library the simulations on particle systems can be easily visualized. Consequently, the study of some physical phenomena like cluster formation, domain walls and vortex patterns in magnetic systems get another dimension. This integration between science and visualization is of paramount importance, either as an efficient debugger or as a tool to understand the simulated physical model.

Acknowledgments

The authors thank Dr. B. V. Costa for valuable discussions during the elaboration of this work.

References

[1] P. A. Fishwick *Web-based Simulation: Some Personal Observations*, Proceedings of the 1996 Winter Simulation Conference, vol. 96, pp.772 -779 1996
[2] C. Monserrata, U. Meierb, M. Alcaizb, F. Chinestac, M.C. Juana *A new approach for the real-time simulation of tissue deformations in surgery simulation*, Computer Methods and Programs in Biomedicine, Volume 64, Issue 2, February 2001, Pages 7785
[3] William Humphrey, Andrew Dalke, Klaus Schulten *VMD: Visual molecular dynamics*, Journal of Molecular Graphics, Volume 14, Issue 1, February 1996, Pages 3338
[4] M.W. Berry, *Massive Data Visualization*, Computing in Science &, Engineering, Vol. 1, No. 4, 1999, pp. 16-17
[5] Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta *Introduction to Parallel Computing*, 2nd ed. ,Pearson Education, Harlow, England.
[6] Peter Pacheco *An Introduction to Parallel Programming*, Elsevier, USA.
[7] Georg Hager, Gerhard Wellein *Introduction to High Performance Computing for Scientists and Engineers (Chapman & Hall/CRC Computational Science)*, CRC Press, Boca Raton, FL.
[8] Barbara Chapman, Gabriele Jost, Ruud van van der Pas *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, Cambridge, Massachusetts.
[9] Eberth Correa, A. B. Lima and B. V. Costa, *Int. J. Mod. Phys.* C, Vol. 23, No. 4 (2012) 1250026.
[10] Dave Shreiner and Bill The Khronos OpenGL ARB Working Group, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Addison-Wesley, Michigan, USA.