Annales UMCS Informatica AI X, 2 (2010) 29-40
DOI: 10.2478/v10065-010-0050-8

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

POSIX threads parallelization for example of Particle-In-Cell density calculations in plasma computer simulations

Anna Sasak*, Marcin Brzuszek

Institute of Computer Science, Maria Curie Sklodowska University,
pl. M. Curie-Sklodowskiej 1, 20-031 Lublin, Poland.

Abstract – The TRQR program [1–4] simulates trajectories of charged particles (electrons or ions) in the electromagnetic field. TRQR is based on the Particle-In-Cell method whose basic guideline is the use of computational particles (called macro particles) that represent a large number of real particles of the same kind moving in the same direction. The program calculates particles charge density distribution and potential distribution for chosen ion sources, analyses particles behaviour in the electromagnetic field, describes the process of beams from the source extraction. A number of factors influences simulation results. In order to improve efficiency the program has been parallelized. This paper presents the process of converting chosen parts of the TRQR program into the multi-thread version. In the first step the program was moved from Fortran 77 to C++. Then it was parallelized using the Pthread library with the standard API for C++ contained in the POSIX IEEE 1003.1c standard. Each of threads has its own stack, set of registers, program counter, individual data, local variables, state information. All threads of particular process share one address space, general signal operations, virtual memory, data, input and output. The Mutex functions were used as a synchronization mechanism. This paper presents the analysis of a particular piece of main program that implements computations of particles density distribution. The paper presents execution time dependencies for different simulation parameters such as: the number of macro particles, size of the simulation mesh and the number of used threads.

1 Introduction

Due to the complexity of physical processes, computer simulations of plasma behaviour in ion sources are still a great challenge for programmers. One of the methods of computing the

*asasak@liza.umcs.lublin.pl

trajectories of charged particles in the electromagnetic field is the Particle-In-Cell method. In the PiC method a large number of particles such as ions or electrons in plasma or beam is represented by a smaller, numerically tractable number of so called 'macro-particles'. Each macro-particle behaves like a single particle of certain kind, but carries a charge large enough to represent all real particles.

This paper presents the results from migration of one piece of TRQR program to parallel mode. First, the program was moved from Fortran 77 to C++ and then parallelized using the Pthread library. The paper presents the results of simulations for different parameters such as a number of used threads, a number of macro particles, mesh size.

2 TRQR - principle of operation

The TRQR program was developed in order to study plasma behaviour as well as the process of extraction and formation of the ion beams emited from the plasma ion sources. The method implemented for computer simulation consists of the following steps:

(1) Setting the systems geometry such as a number of particles etc. and generating initial distribution for all kinds of particles.
(2) Calculations of particles density distributions for chosen ion sources using the PiC method.
(3) Solving the Poisson equation for the charge density obtained in the previous step and the boundary conditions imposed by electrodes.
(4) Calculation of electrical field in the grid points.
(5) Solving the Lorentz equations of motion for each particle.
(6) Generating new particles if it is needed due to hits on electrodes and plasma chamber walls.

This procedure, steps from 2 to 6, continues until a final state is achieved[3].

The special subject of interest for this paper is the particle-in-cell (PiC) method the second step of simulation is based on. In the PiC method a large number of particles such as ions or electrons in plasma or beam is represented by a smaller, numerically tractable number of so called 'macro particles'. Each macro particle behaves like a single particle of certain kind, but carries a charge large enough to represent all real particles. The simulation space is divided into small regions creating a spatial mesh. The method weights particles to grid points using a particle shape factor to obtain charge on the grid. This distribution process is carried out with one of two possible schemes. The first method called nearest grid point (NGP) assigns the macro-particle charge to the point of grid that is the nearest to the particles position. In the second one called cloud-in-cell (CiC) fractions of macro-particle charge are assigned to 8 (in the case of 3D calculations) nearest in the mesh grid points. Even better charge distribution is obtained if in the CiC method the macro particle charge is distributed among 27 nearest grid points [4].

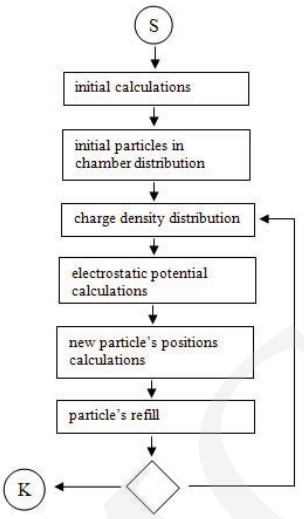


Fig. 1. Block scheme for the TRQR program.

3 POSIX threads API

In architectures with shared memory threads can be used to implement parallelism. For the Unix systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. The already mentioned POSIX standard from 1995 is included in the Unix system distributions.

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. The comparison between threads and processes is presented in Table 1.

What needs to be emphasized is that in the case of threads - reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

The subroutines which comprise the Pthreads API can be informally grouped into three major classes (included in the library Pthreads):

(1) Thread management – the group of functions that work directly on threads - creating, detaching, joining, etc. Here are also included the functions that set thread attributes.
(2) Mutexes (abbreviation for 'mutual execution') – the functions that deal with synchronization. The Mutex functions provide for creating, destroying, locking and unlocking mutexes and also setting or modifying mutex attributes.
(3) Condition variables – the functions that address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes the functions to create, destroy, wait and signal based upon specified variable values. In this paper condition variables are only mentioned without further analysis as they were not implemented in the pthread parallelization presented in this paper.

Table 1. Process and thread features comparison.

PROCESS	**THREAD**
• Created by the operating system • Requires a fair amount of overhead • Contains information about program resources and program execution state that include: – Process, process group, user and group IDs, – environment, – working directory, – program instructions, – registers, – stack, – heap, – file descriptors, – signal actions, – shared libraries, – inter-process communication tools.	• Use and exist within the process-creator resources • Duplicate only the bare essential resources that enable them to exist as executable code • Share with other threads in the same process: – Global and static variables, – heap and dynamic variables (Two pointers having the same value point to the same data), – operating system resources (files), – process instructions. • Each thread has a unique: – Set of registers, stack pointer, – automatic variables, – Stack for local variables, – priority, – thread ID.

4 Thread creation

Initially *main()* program comprises a single thread. All other threads must be created explicitly by the programmer. Once created, threads are peers and may create other threads. There is no implied hierarchy or dependency between them. A new thread is created by calling *int* ***pthread_create****(pthread* ****thread****, const pthread_attr* ****attr****, void *(**start_routine**)(void *), void **arg**)* subroutine. The arguments of this function in order of appearance stand for: unique identifier for the new thread returned by the subroutine, attribute object that may be used to set thread attributes, the C routine that will be executed by thread once it is created, a single argument that may be passed to *start_routine*. Attribute parameter set to NULL means that default attributes are used, otherwise it defines members of struct *pthread_attr_t* that includes: detached state, scheduling policy, stack address and size etc. As it was mentioned before *pthread_create()* routine permits a programmer to pass only one argument to the thread start routine. To overcome this limitation a structure should be created which contains all of the arguments to be passed. Then just a pointer to that structure should be passed to *pthread_create()* routine.

There is presented below the fragment of code, which creates NTH threads with a default set of parameters which will execute routine *thread_fun_dens* with the parameters from the proper cell of matrix *tab_th_data*.

```
struct th_data{
  long idoms;//starting cell of global density matrix
  long idome;//ending cell of global density matrix
  long NNion;//number of ions per thread
};
pthread_t th_ids[NTH]; //matrix that contains threads ids
th_data tab_th_data[NTH];//matrix of threads specific data,
    passed as a structure pointer to the executed routine

void *thread_func_dens(void *ptr) {
  ...
  pthread_exit(NULL);
}

void main(...) {
  ...
  for (int w=0; w< NTH; w++)
    pthread_create( &th_ids[w], NULL, thread_func_dens, (void
        *)&tab_th_data[w]);
  ...
}
```

5 Threads synchronization and termination

There are several ways in which a thread may be terminated. The most common is either when the thread returns from its starting routine or when the thread makes call to the *pthread_exit()* subroutine. Typically, the *pthread_exit()* routine is called after a thread has completed its work and is no longer required to exist. If main() finishes before the threads it has created, and exits with *pthread_exit()*, the other threads will continue to execute. Otherwise, they will be automatically terminated when *main()* finishes. The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.

One way to accomplish synchronization between threads is so called 'joining'. The *int* ***pthread_join****(pthread_t* ***th****, void ********thread_return****)* subroutine blocks the calling thread until the thread specified by *th* argument terminates. The programmer is able to obtain, via the second argument, the target threads termination status. It is possible though only if it was explicitly specified in the target thread call to *pthread_exit* routine. A joining thread can match

only one *pthread_join()* call. It is a logical error to attempt multiple joins on the same thread. In the following figure the scheme of program course is presented, which after creating two worker threads waits for them to exit and then resumes its execution.

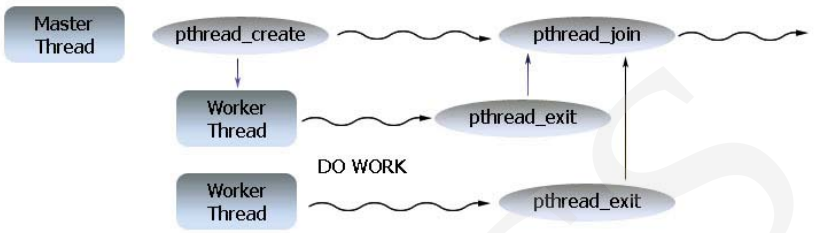


Fig. 2. Threads synchronization.

The fragment of main function that stops program execution until all created threads exit would have the following form:

```
void main(...) {
  ...
  for (int ii=0; ii< NTH; ii++)
          pthread_join( th_ids[ii], NULL); //execute as much
              pthread_joins as pthread_create
              //were execute before
  ...
}
```

6 Mutual execution

Mutex variables are one of primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. A mutex variable acts as a 'lock' or a semaphore protecting access to a shared data resource – critical section. With the basic mutex concept only one thread can own – which means lock – a mutex variable at any given time. Thus, even if several threads try to lock a certain mutex only one of them will succeed, booking access to the protected resource for himself. The shared data resource is available again not till then mutex owner unlocks that mutex. The presented operation is a safe way to ensure that when several threads update the same variable, the final value is the same as what it would be if only one thread performed the update.

The typical sequence of steps in the use of a mutex is as follows:

(1) a mutex variable is created and initialized,
(2) several threads attempt to lock the mutex,
(3) only one of them succeeds and that thread owns the mutex,
(4) the owner thread performs a set of actions,

(5) the owner unlocks the mutex,
(6) another thread acquires the mutex and repeats the process,
(7) finally the mutex is destroyed.

The mutex variable must be declared with the type *pthread_mutex_t* and initialized before it can be used. Initialization can take two forms:

(1) static with the instruction
pthread_mutex_t mutex = ***PTHREAD_MUTEX_INITIALIZER***,
(2) dynamic with *int* ***pthread_mutex_init****(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)* routine.

Initially mutex is unlocked. To establish different from default (specified as NULL) properties for the mutex the second argument of the pthread_mutex_init routine should be used. Mutex that is no longer needed should be released with ***pthread_mutex_destroy****(pthread_mutex_t *mutex)* routine.

Three standard routines are used to manage mutex access. The ***pthread_mutex_lock****(pthread_mutex_t *mutex)* routine is used to acquire lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked. The ***pthread_mutex_trylock****(pthread_mutex_t *mutex)* will attempt to lock a mutex. However, if the mutex is already locked, the routine will return with 'busy' error code. The ***pthread_mutex_unlock****(pthread_mutex_t *mutex)* will unlock a mutex if called by owning thread. An error will be returned if the mutex has already been unlocked or if the mutex is owned by another thread[5].

The following example presents the way mutexes were used in our simulation.

```
pthread_mutex_t ***tab_mutex;
...
for (int x=1; x<=Nxx; x++)
  for (int y=1; y<=Nyy; y++)
    for (int z=1; z<=Nzz; z++) {
      int res = pthread_mutex_init(&tab_mutex[x][y][z],NULL);
    }
...
//creating threads with pthread_init routine
...
        //a piece of code somewhere in the thread
            start_routine
        int err = pthread_mutex_lock( &tab_mutex[Nx][Ny][Nz] )
            ;
        density_q[Nx][Ny][Nz][kj] += is;
        int err2 = pthread_mutex_unlock( &tab_mutex[Nx][Ny][Nz
            ] );
...
```

```
for (int x=1; x<=Nxx; x++)
  for (int y=1; y<=Nyy; y++)
    for (int z=1; z<=Nzz; z++) {
      int res=pthread_mutex_destroy(&tab_mutex[x][y][z]);
    }
```

7 Parallel mode calculations

The environment for simulations was the Intel Xeon processor 4cores x 2, 16BG RAM, Mandriva operating system and gcc 4.1.2 compiler. In the first step the program was moved from Fortran 77 to C++. Then it was parallelized using the Pthread library with the standard API for C++ contained in the POSIX IEEE 1003.1c standard.

During the simulation process the measure that was analysed was the simulation time. It is a formal but very relative measure as sometimes the process of creating parallel version may not be cost effective contrary to the gained reduction in the simulation time. The second performance criterion that was adopted for plasma density thread parallelization is speedup that is described by the formula $S(p) = \frac{T(1)}{T(p)}$, where p stands for a number of threads, $T(1)$ and $T(p)$ - the simulation time with one or p threads (adequately) [6].

8 Results of simulations

As it was presented in paper [7] using the simplest charge density distribution technique and a large number of macro particles is the best solution as far as charge density calculations are concerned. For example, using NGP and 100 mill of macro particles gives better results (i.e. more homogeneous distributions) in less time than using the CIC method and 20 mill of macro particles. That is why all results presented in this paper are calculated for the NGP method with a different number of macro particles, different sizes of spatial mesh and a different number of threads used in the parallelization process.

Fig. 3 presents the simulation time for the NGP method with different numbers of macro particles and the mesh of size 100x100x100. Red line in each picture stands for the execution time of the sequential version of the algorithm.

Analyzing the above graphs one can conclude that using only two threads gives the execution time close to the sequential version and that using eight threads, which equals the number of available processors, gives the best reduction of execution time. Further improvement of a number of threads, nine and above does not give further reduction of execution time.

As the graphs obtained for simulations with a different number of macro particles show similar results, Fig. 4 presents speedup calculated only for one of them, the one for 200 mill macro particles. It confirms that speedup close to 1 (which means close to the sequential execution time) is for 2 threads and the highest speedup is gained for 8 threads.

In the next step the size of the mesh was changed to 50x50x50. Two simulations were done. First for 200mill of macro particles – Fig. 5(a). In the second one – Fig. 5(b) - the number

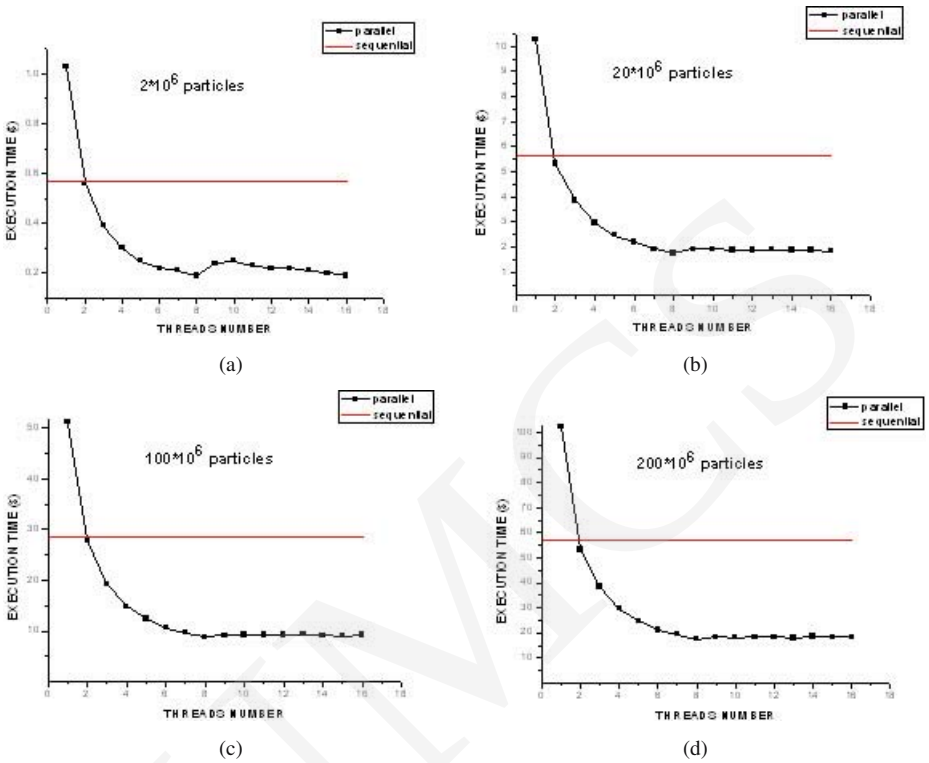


Fig. 3. Time of charge density calculations as a function of the number of threads used for the parallel run, using the NGP method, mesh of size 100x100x100 and a different number of macro particles: a)2mill, b)20mill, c)100mill, d)200mill.

of particles was changed proportionally to the change in mesh size, which gave the number of approximately 25mill macro particles. For both simulations speedup factors were calculated and presented in Fig. 6(a) and 6(b) respectively.

Analyzing Fig. 5 and 6 it can be noticed that the maximum speedup gained with the parallelization changed dropped by about 40% compared to the previous simulation. Also the number of threads required to gain the execution time close to sequential changed from 2 to 4.

Further tests were carried out for different sizes of mesh from 200x200x200 down to 15x15x15. For each of them the parallel version run for 200mill macro particles and 8 threads were executing calculations. The red line stands for the execution of sequential version of the algorithm.

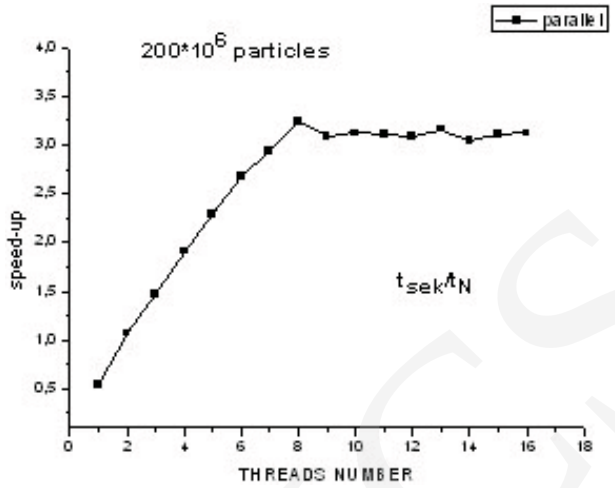
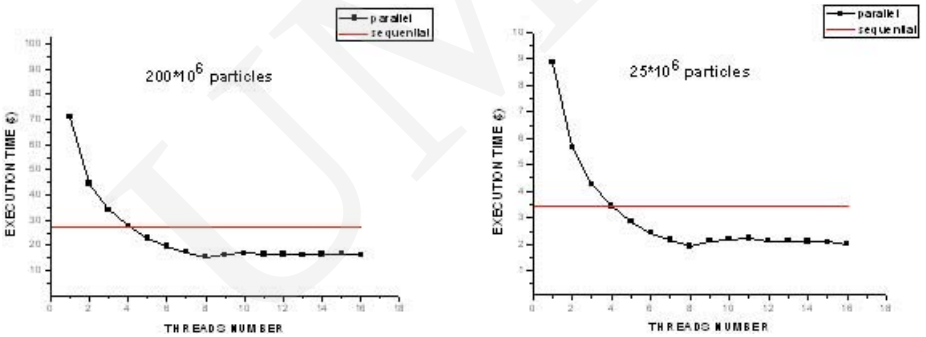


Fig. 4. Speedup for NGP parallel run, for 200mln. macro particles mesh of size 100x100x100.



(a) (b)

Fig. 5. Time of charge density calculations versus the number of threads used for the parallel run, using the NGP method, mesh of size 50x50x50 and different number of macro particles: a)200mill, b)25mill.

Fig. 7 presents that for the meshes of size 80x80x80 and bigger ones give quite good execution time reduction while parallelized. In the case of meshes of size 40x40x40 and smaller running the parallel version of algorithm gives no benefit of reduction of execution time.

Final tests were carried out for the asymmetrical mesh of dimensions 128x64x128 and 100 mill macro particles. The aim of this test was to examine if the geometry of the mesh has any influence on the algorithm performance. Fig. 8 presents the results of that simulation – both simulation time and speedup. The environment of this simulation is similar to the one presented in Fig. 3(c). The results for both mentioned simulations are very close which gives

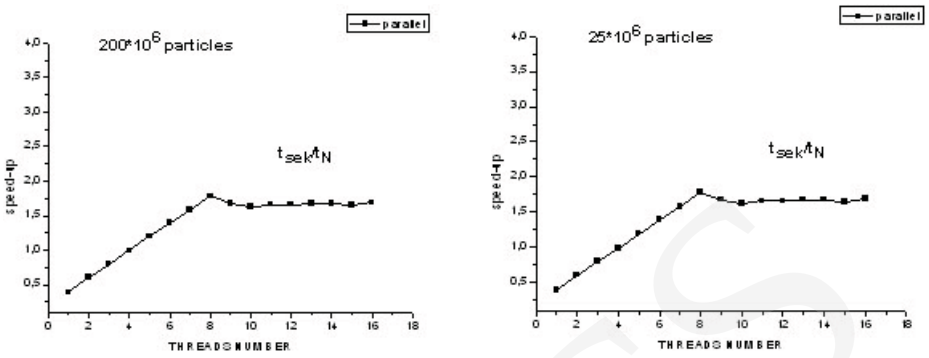


Fig. 6. Speedup for NGP parallel run, for a) 200mill. b) 25mill. macro particles, mesh size 50x50x50.

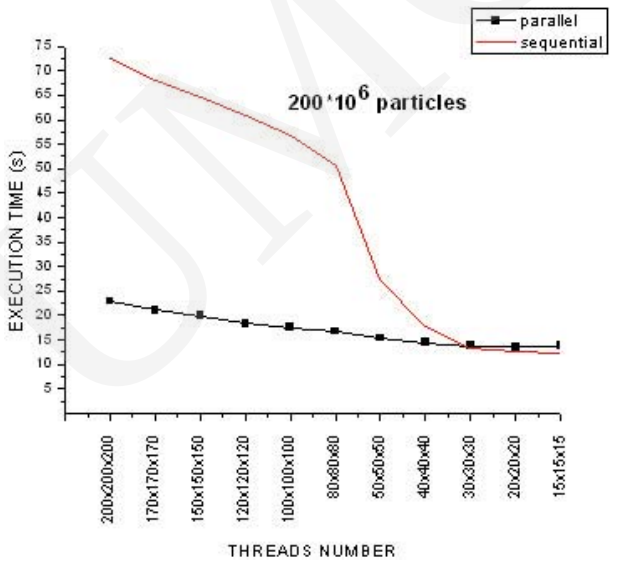


Fig. 7. Time of charge density calculations versus the mesh size with the number of threads used for the parallel run equal 8, using the NGP method and 200mln. of macro particles.

a conclusion that only a number of cells influences the simulation time whereas the mesh geometry has no influence on POSIX thread parallelization performance.

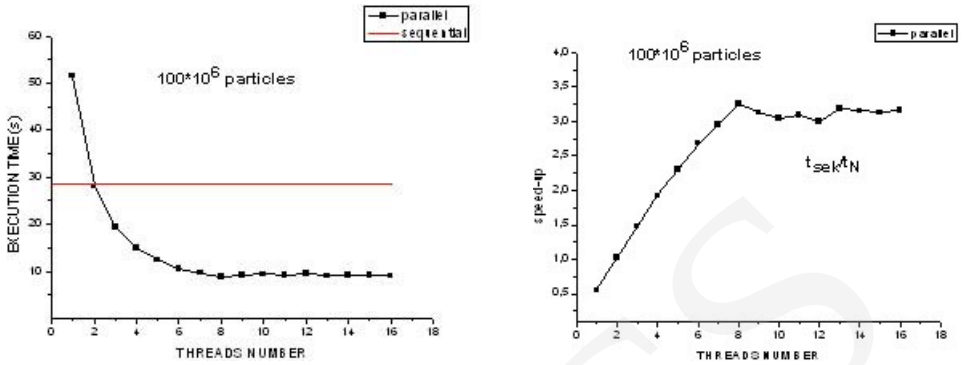


Fig. 8. Time of charge density calculations and speedup for the parallel run, using the NGP method, the mesh of size 128x64x128 and 100mill. of macro particles.

9 Conclusion

A direct advantage of program parallelization is more effective time use which relates to the time assigned to the simulation process. This paper presents the POSIX Pthread library as one of the available methods of parallelization. So far Pthread parallelization is implemented only for a part of TRQR program which is charge density calculations, but it gives quite acceptable results encouraging for further research.

References

[1] Staebler A., Sielanko J., Goetz S., Speth E., Fusion technology, Elsevier Science & Technology 26(2) (1994): 145.

[2] Sielanko J., Muszyński M., Electron technology 30(4) (1997): 352.

[3] Sielanko J., Turek M., Tanga A., Computer simulation of the potential distribution inside the plasma chamber of negative ion source, Annales UMCS Informatica 2 (2004): 251–262.

[4] Turek M., Droździel A., Pyszniak K., Sielanko J., Extraction of the ion beam from hollow cathode ion source. Experiment and computer simulation (Vacuum, 2005): 649–654.

[5] Nichols B., Buttlar D., Proulx F., Pthreads programming. A POSIX standard for better multiprocessing (O'Reilly, 1996): 61–98.

[6] Wilkinson B., Allen M., Parallel programming. Techniques and applications using networked workstations and parallel computers (Prentice Hall, 2004): 61–71.

[7] Brzuszek M., Turek M., Sielanko J., Parallelization of the 'Particle-in-Cell' (PIC) density calculations in plasma computer simulations, Annales UMCS Informatica 6 (2007).