S E S 2 0 1 3
Ninth Scientific Conference with International Participation
SPACE, ECOLOGY, SAFETY
20 – 22 November 2013, Sofia, Bulgaria

PARALLEL SOLVING OF SITUATIONAL PROBLEMS FOR SPACE MISSION ANALYSIS AND DESIGN

Atanas Atanassov

Space Research and Technology Institute – Bulgarian Academy of Sciences
e-mail: At_M_Atanassov@yahoo.com

Keywords: *situational analysis, orbital events prediction, parallel calculations, parallel algorithms, multi-core calculations.*

Abstract: *The preparations of space missions are based on different kinds of simulations. The design of satellite missions demands taking in account many geometrical and physical constrains. The determination of time windows of satellite orbital events (suitable for satellite experiments or routine operations) based on computer simulation and specific algorithms is known as* ***situational analysis****. When complex multi-satellite (with multi sensors on every of them) missions are analyzed, many such situation problems are solved for every sensor or complex of one type or different type of sensors. Some of situational problem account more than one constrains and some of them are connected with many calculations.*

One way for reducing of calculation time for situational problems is parallelization of algorithms and using of multi core processors. An idea for situational analysis problems solver based on threads' parallelism is described. Some fragments of program realization in Fortran 95 language are shown. This solver is intended to work as subsystem of system for analysis and design of space multi-satellite missions. The solver is developed for working on multi core platforms.

ПАРАЛЕЛНО РЕШАВАНЕ НА СИТУАЦИОННИ ЗАДАЧИ ПРИ АНАЛИЗ И ПРОЕКТИРАНЕ НА КОСМИЧЕСКИ ЕКСПЕРИМЕНТИ

Атанас Атанасов

Институт за космически изследвания и технологии – Българска академия на науките
e-mail: At_M_Atanassov@yahoo.com

Резюме*: Подготовката на космически експерименти се основава на различни видове симулации. Проектирането на космически експерименти изисква отчитане на геометрични и физически ограничения. Определяне на времеви интервали подходящи за провеждане на експерименти и извършване на рутинни операции, основано на компютърни симулации и специални алгоритми е известно като* ***ситуационен анализ****. При анализ на многоспътникови експерименти (с много уреди на всеки от спътниците) ситуационни задачи се решават за всеки уред поотделно или за уреди от еднакви или различни типове, участващи в решаването на една или повече научни задачи. Някои от ситуационните задачи са свързани с проверка на повече от едно условия, които могат да изискват много изчисления.*

Начин за намаляване на изчислителното време за ситуационни задачи е паралелизация на алгоритмите и използване на многоядрени процесори. Описана е идея за процесор за решаване на ситуационни задачи основан на изчислителни нишки. Показани са фрагменти и подпрограми на Fortran 95. Този процесор е замислен да работи като подсистема на програмна система за симулации на многоспътникови експерименти.

Introduction

The simulation of complex satellite missions, related to a broad circle of scientific and applicable problems, based on different kind of experiments and determination of suitable condition for their performance, demands satisfaction of large number of temporal, angle, spatial, physical and other kind of restrictions. Such restrictions are fulfilled on time intervals, while the satellites move by their orbits, depending on numerous different parameters of sensors and other instruments, parameters of environment (geo-magnetic and electric fields, radiation background), visibilities of the

Sun and other sky objects. The satisfactions of such restrictions are known as orbital events [1]. The time intervals mentioned above are known as “time windows” and play very important role in theory and practice of planning and scheduling space missions [2,3]. Determination of suitable for performance of space experiments time windows is very important on different stages of missions design.

The analysis for determination of time windows for every scientific problem is known as situational analysis (SA) [4]. Many scientific and practical problems lead to many situational problems (SP) to be solved, which require significant calculation time.

A model was presented [5] for description of situational conditions (SC) and SP, which includes more than one SC and processor for their solution.

Paraphrasing Simonsen [6] we can point several reasons the solution of time windows prediction problems related to space missions and experiment simulations to require much calculation time:

- complex calculation models are used for the calculation of different constraints of the events;
- simultaneous solving of many time windows prognostication problems are included in a simulation model;
- the investigated time interval (t_0,t_{end}) is large;
- repeatedly solving of events prediction problem for determination of the optimal space mission parameters.

Solving of large number SP in the frame of one space mission simulation could be related to much overhead and gives possibilities to search optimal solution. One way for acceleration of the calculation process is connected with applying of parallel calculations.

Possible ways for parallelization of situational problems solving

Two basic approaches for achieving parallelism in broad circle of problems are possible:

- parallelism across the method;
- parallelism across the system.

In the case of SA the first approach suppose a presence of possibilities for parallelization in the frames of the particular situational conditions. This is possible when particular independent segments present in the code, which could be executed in parallel on different processor cores (pipe-lining), or for some kinds of cycles with independent iterations.

“Parallelization across the system” is approach which is applicable to complex calculation model which could be decomposed on independent parts and every one of them could be executed as particular thread. This approach is used exclusively in the present work.

The second approach contains more potential in cases of SP, because the different conditions in the frame of one SP could be calculated on different processor core. When we have many SP then every one of them is possible to be solved on different processor.

Situational analysis processors parallelization approach

A processor for solving of situational analysis problem (SAP) was proposed in [5]. It was based on:

- model for description of elementary situational condition and situational problems;
- set of subroutines for every situational condition.

The processor receives set of preliminary defined situational problems and executes them in sequential steps of the system time. The different situational problems can be composed from different number of situational conditions. Every situational condition is described with specific set of parameters. The time for verification of every situational condition and the times for solving of different situational problems are different.

The new parallel version of SAP – parallel situational problem solver (PSPS) is proposed in the present paper. It is based on “pool of threads” program model [7]. This model is based on portioning of entire problem to particular sub-problems. Every sub-problem is one situational problem in our case. The number of solved situational problems could be much larger than the number of threads of the pool. The number of threads must be equal or less than the number of processor cores. When one thread finishes the execution of one situational problem it attempts to get a new one if there are such. In contrast to classical case, here threads compete in the frame of every system time step. Additional synchronization is necessary between threads and parent thread.

Program realization

The creation of “pool of threads” is made on appropriate place in the program by special subroutine **CreateThreadsPool**. This is the same subroutine, which is used in our previous studies, for initialization of ordinary differential equation systems [8], but now it is made universal and will be applied in all analogous cases (fig. 1 and fig. 2).

```
external                          SituationProcessor
…
   CALL  CreatePoolThreads(SituationProcessor,num_Sit_threads,Sit_thread_par,Sit_ha_1)
                  ……..
   CALL  Data_Sit_Solver(num_sat,t,dt,xvn,xvk,dTrajectoryParam,sci_task,max_num_sit, &
                                                                          num_sci_task)
   CALL  Preparation_Sit_Solver(num_Sit_threads,Sit_thread_par,Sit_ha_1)
…
 DO   …
…
   CALL  sit_prob (num_sat,t,dt,xvn,xvk,dTrajectoryParam, &
                          sci_task,max_num_sit,num_sci_task) !
…
 END DO
```

Figure 1. Code fragment illustrating creation and initialization of situational analysis solver. The subroutine ‘Data_Sit_Solver’ prepares data for transferring to solver; the subroutine ‘Preparation_Sit_Solver’ executes control functions.

```
ENTRY    Preparation_Sit_Solver(num_SitThreads,thread_par,Sit_ha_1)

ALLOCATE (ha_end(num_SitThreads),     ha_beg(num_SitThreads),STAT=isv2);
                ha_end(:)= thread_par(4,:); ha_beg(:)= thread_par(3,:);
          ALLOCATE (ha    (num_SitThreads));    ha    (:)= thread_par(1,:)
         ha_1= Sit_ha_1;         num_Sit_threads= num_SitThreads
                  DO  i=1,num_SitThreads
                    k1= ResetEvent      (ha_beg(i))
                    k2= ResumeThread(ha       (i))
                  END DO; num_Sit_threads= num_SitThreads; ha_1= Sit_ha_1
          Sit_thread_par_adr= LOC(thread_par) ! Sit_thread_par_adr is transfered to buffer
                                              ! subroutine by common area

RETURN
ENTRY    Data_Sit_Solver(num_sat,t,dt,xvn,xvk,dTrajectoryParam,sci_task, &
                                                 max_num_sit,num_sci_task)

      numsat= num_sat; num_sit= max_num_sit; num_task= num_sci_task
             t_adr= LOC(t);   xvn_adr= LOC(xvn); TrajectParam_adr= LOC(dTrajectoryParam)
           dt_adr= LOC(dt); xvk_adr= LOC(xvk);   sci_problem_adr= LOC(sci_task)
       glb_counter= 0
  adr_glb_counter= LOC(glb_counter)

RETURN
```

Figure 2. Code fragments of subroutine ‘Data_Sit_Solver’ and ‘Preparation_Sit_Solver’

The subroutine, which creates the solver, starts some number of threads. The subroutine creates also an event **ha_1** which is used for synchronization and control of the threads by getting next situational problem (fig. 5) and ensures that there is no doubling and solving the same problem from more than one thread. A couple of events are created for every thread, which serve for synchronization of their control. One of the events **ha_beg** (fig. 3, fig. 5) starts execution of every of the threads for every step of the system time. Every thread report that the calculations are finished when there are not more situational problem for solving by using the second event **ha_end** (fig. 3, fig. 5).

A buffer subroutine **SituationSolver** is used as parameter for creation of threads of the situational solver. Addresses of all data for processing and control are transferred to PSPS through this subroutine. Global type data are used for transfer of these addresses on this stage.

```
SUBROUTINE     sit_prob(num_sat,t,dt,xvn,xvk,eps)
 USE  DFmt
  real*8                t,dt,xvn(6,num_sat),xvk(6,num_sat),eps(6,num_sat)
  common   /cadr_traekt/adr1,adr2,glb_counter,numsat
… … …
          glb_counter= 0;
a: DO  i=1,num_thrd
      k= SetEvent    (ha_beg(i)) ! Events for start of threads
  END DO a
      k= WaitForMultipleObjects(num_thrd, ha_end,WaitAll,Wait_infinite)
b: DO  i=1,num_thrd
      k= ResetEvent(ha_end(i)) ! Events for waiting ending of threads
  END DO b
… … …
END SUBROUTINE  sitanal
```

Figure 3. The **sit_prob** is subroutine which serves for synchronization between parent thread and threads of the solver and for transfer of control data and results between parent thread and threads of the solver.

```
SUBROUTINE     SituationSolver(th_id_num)
 USE DFlib
 USE DFmt
   integer                              th_id_num
!___Transferred pool control parameters_____________________
   integer                  Sit_thread_par_adr, ha_1
   common   /cSit_nth/   num_Sit_threads,Sit_thread_par_adr, ha_1 ! nth- threads number
   common   /cSit_const/num_sat,max_num_sit,num_sit_prob  ! constant data to buffer
subroutine
   integer          t_adr,dt_adr,xvn_adr,xvk_adr,TrajectParam_adr,sci_problem_adr
   common   /cSit_data/ t_adr,dt_adr,xvn_adr,xvk_adr,TrajectParam_adr,sci_problem_adr
   integer                    adr_glb_counter,glb_counter,ha_1l
   common   /cSitThrdPool/adr_glb_counter
!___________________________________________________________
   integer     thread_par_local(4),thread_par(4,num_Sit_threads) !
   AUTOMATIC   ha_1l
       POINTER(Sit_thread_par_adr,thread_par)

        ha_1l= ha_1 ! locat value of the handler
               thread_par_local(:)= thread_par(:,th_id_num);   ! Storing tread's parameters
                                                               ! in the tread's stack storage

 CALL  SitProblemsPool(num_sat,t_adr,dt_adr,xvn_adr,xvk_adr,TrajectParam_adr, &
          sci_problem_adr,max_num_sit,num_sit_prob,thread_par_local,adr_glb_counter,ha_1l)

END SUBROUTINE  SituationSolver
```

Figure 4. The **SituationSolver** is buffer subroutine which serves to transfer initial control data to solver.

The threads are created in suspended state. When all variables and areas necessary for initialization of PSPS are loaded with information, the state of the threads is changed to non-suspended and their execution begins. Every thread copies transferred information in private storage and reaches place in the solver were waits for beginning of the real work connected with solving of situational problems.

Instead of standard calling of particular subroutine, the control of the threads is realized by changing the states of the control events of the thread in **sit_prob** subroutine, which is part from the parent thread. This is possible by respective library functions **SetEvent** and **ResetEvent** [9]. The parent program starts the threads on appropriate place and transits in waiting state until all situational problems are solved. A program fragment which illustrates this is shown on figure 3.

Figure 4 presents the buffer subroutine **SituationSolver**. All control parameters and data are transferred through common areas.

```
SUBROUTINE     SitProblemsPool(numsat,t_adr,dt_adr,xvn_adr,xvk_adr,TrajectParam_adr, &
                    sci_problem_adr, max_num_sit,num_sit_prob,thread_par,adr_glb_counter,ha_1)
USE DFmt
 USE  RN
  integer                    t_adr,dt_adr,xvn_adr,xvk_adr,TrajectParam_adr,sci_problem_adr, &
                        thread_par(4),adr_glb_counter,ha_1
!
  type    (TrajectoryParam)    TrajectParam(numsat)
  type    (sit_task)               sci_task(0:max_num_sit,num_sit_prob) !
  integer            glb_counter
  integer, AUTOMATIC :: loc_ha_1,ha_beg,ha_end, loc_numsat !
  real*8             t,dt,xvn(6,numsat),xvk(6,numsat)
   POINTER(t_adr,t);          POINTER(dt_adr,dt)
   POINTER(xvn_adr,xvn);   POINTER(xvk_adr,xvk); POINTER(sci_problem_adr,sci_task)
   POINTER(adr_glb_counter,glb_counter);              POINTER(TrajectParam_adr,TrajectParam)

        ha_beg= thread_par(3);      ha_end= thread_par(4); ! Stored in the threads' stack storage
       loc_ha_1= ha_1;            loc_numsat= numsat;

 DO WHILE(.true.)
    k=  WaitForSingleObject(ha_beg,WAIT_INFINITE)  ! Event for thread's starting for time step
  DO WHILE(glb_counter.LT. num_sit_prob)                    !
    k= WaitForSingleObject(loc_ha_1,WAIT_INFINITE); ! The event 'ha_1' preserves doubling
                                                          ! of situation problem
                                                    ! until current local pointer is formed 'ha_1'is
                                                    ! made un-signaled automatically
              glb_counter= glb_counter + 1;     ! Toward the next situation problem
         loc_counter= glb_counter;              ! Storing in local thread's storage for using as
pointer
                  k= SetEvent(loc_ha_1)     ! Allow other thread to may select situation problem
                                            ! if there is
           IF(loc_counter.GT. num_sit_prob) EXIT

   CALL  Psitanal(loc_numsat,t,dt,xvn,xvk,TrajectParam,sci_task(0,loc_counter),max_num_sit)

  END DO;!
    k= ResetEvent(ha_beg) ! Prepare the threads' starting event for the next time step
    k=    SetEvent(ha_end) ! Signals parent for end of solver threads' calculations
 END DO;

END SUBROUTINE  SitProblemsPool
```

Figure 5. **SitProblemsPool** subroutine organizes the synchronizations of the threads with main routine in the frame of the pool and calls the real situational processor (subroutine **Psitanal**).

Figure 5 shows the code of the subroutine **SitProblemsPool**, which illustrates the joint work of the threads in the frame of the pool and all communication processes with the main program. Every thread of PSPS calls subroutine **Psitanal** which solves only one SP. The subroutine **Psitanal** is version of previous serial code subroutine **sitanal** [5]**.**

Conclusion and future work

A stage of development of PSPS, which determines time intervals appropriate for performing measurements in the frames of space missions, is proposed. PSPS is intended for parallel calculations on platforms with multi-core processors and shared memory systems. At this stage it is included in developed program system for simulation of multi-satellite space experiments [5].

Experiments will be performed for establishment of effectiveness of PSPS by competition with other parallel subsystem in the frames of above pointed program system. On this stage, the integrator of ordinary differential equation system is the basic competitor for using processor cores. This competition is not fully direct because the two subsystems don't work simultaneous – the results from the integrator are used from the situational solver.

Development of algorithms and codes for new situational conditions are in progress.

References:

1. L e e y, B.-S. and K i m, J.-H., Design and Implementation of the Mission Planning Functions for the KOMPSAT-2 Mission Control Element, J. Astron. Space Sci. 20(3), 227–238 (2003)
2. P e m b e r t o n, J. C., F. A G a l i b e r, constraint-based approach to satellite scheduling. In E.C. Freuder, R.J.Wallace (editors), Constraint Programming and Large Scale Discrete Optimization, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 57, pages 101-114, 1998.
3. H a r r i s o n, S. A., M. E., P r i c e, Task scheduling for satellite based imagery. In Proce. of the Eighteenth Workshop of the UK Planning and Scheduling Special Interest Group, pages 64-78, University of Salford, UK, December 1999.
4. P r o k h o r e n k o, V. I., Study of satellite situations mission. Acta Astronautica, v.10 №7, 1983, 499-503.
5. A t a n a s s o v, A., Program System for Space Missions Simulation - First Stages of Projecting and Realization, Proceedings of "Eighth scientific conference - Space Ecology Safety", 2012, 209-214.
6. S i m o n s e n, H. H. Exrapolation methods for ODE's: continuous approximations, a parallel approach, Ph. D. thesis, Math. Sci. Div., Norwegian Inst. Of Tech., Trondheim, Norway, 1990.
7. R a u b e r, T. and R ü n g e r G., Parallel Programming. For Multicore and Cluster Systems, Springer, 2010, 455.
8. Digital Visual Fortran Programmer's Guide.
9. A t a n a s s o v, A., An Adaptive Parallel Integrator of Ordinary Differential Equations System for Space Experiment Simulation, 2012, 203-208.