UNIVERSITY OF MICHIGAN

KIVA-3Vp: A Parallel Patch

EECS-587: Final Project

Gandharv Kashinath

12/20/2009

Abstract

As part of the term project for EECS 587-Parallel Computing, an attempt was made to improve the performance of the research and industry code KIVA. In order to accomplish this task, the limiting chemistry modeling segments of the code were parallelized using MPI. The parallel capabilities were evaluated for speed and efficiency and significant overall improvement in the algorithm performance was demonstrated. The main emphasis of this project was to efficiently parallelize the algorithm; hence details on the physics of the problem being solved have been abbreviated. In the following sections, a brief overview of the KIVA setup followed by parallelization approach and results on performance analysis are presented and analyzed.

1. Introduction

1.1 KIVA – Internal Combustion Engine Physics Solver

KIVA is a transient, three-dimensional, multiphase, multicomponent code (FORTRAN 77) for the analysis of chemically reacting flows with sprays has been under development at the Los Alamos National Laboratory for the past several years [3]. Current version used by our research group is a serial code where the Navier- Stokes equations are solved and in order to complete the combustion modeling (source term), the commercial software CHEMKIN is called to do the finite rate chemistry. This code has been extensively applied to solve Internal Combustion (IC) engine problems where the complete physics of chemically reacting flows needs to be solved dynamically.

1.2 Motivation

The serial code currently being implemented is slow and significant improvements in performance can be obtained by introducing some parallelization into the algorithm. The structure of the code is very well suited to parallelize the segments of the code limiting the performance namely, the chemistry solve performed by CHEMKIN. The chemistry subroutine is highly decoupled from the entire system and hence can be relatively easily parallelized using a “modified” master-slave approach in MPI.

1.3 Message Passing Interface (MPI) – Parallelization Tool

The Message Passing Interface, or MPI, is a standard library of subroutines (FORTRAN) or function calls (C) that can be used to implement a message passing program. MPI allows for the coordination of a program running as multiple processes in a distributed memory environment. It also allows efficient implementations across a range of architectures, offers a great deal of functionality, including a number of different types of communication and special routines for common *collective* operations, the ability to handle user-defined data types and topologies, and support for heterogeneous parallel architectures. These features of MPI make it an ideal candidate to help implement the parallelization of KIVA and thus improve its overall performance [7].

2. Overview of KIVA – Need for Parallelization

KIVA is a transient, three-dimensional, multiphase, multicomponent code for the analysis of chemically reacting flows with sprays written in FORTRAN 77. The code uses an Arbitrary Lagrangian Eulerian (ALE) methodology on a staggered grid, and discretizes space using the finite-volume technique. The code uses an implicit time-advancement with the exception of the advective terms that are cast in an explicit but second-order monotonicity-preserving manner. Also, the convection calculations can be subcycled in the desired regions to avoid restricting the time step due to Courant conditions [1]. The range of validity of the code extends from low speeds to supersonic flows for both laminar and turbulent regimes. Arbitrary numbers of species and chemical reactions are allowed. A stochastic particle method is used to calculate evaporating liquid sprays, including the effects of droplet collisions and aerodynamic breakups. Although specifically designed for performing internal combustion engine calculations, the modularity of the code allows it for easy modifications for solving a variety of hydrodynamics problems involving chemical reactions. The code has found a widespread application in the automotive industry. For a detailed consideration of the mathematical model and numerical consideration please refer to D' Ambra [2].

The performance of this code is highly limited by the call to CHEMKIN which is a tool for incorporating complex chemical kinetics into simulations of reacting flow. The reaction kinetics of the problem is modeled as a chemical source term in the Navier-Stokes equations that are being solved by KIVA. In order to obtain this chemical source term a series of ordinary differential equations (ODE) need to be solved throughout the entire domain of the problem. Though the ODE solver implemented by CHEMKIN is highly optimized for performance, the call to this module is done in a serial fashion i.e. one grid location at a time. By parallelizing this call where all the grid points perform this operation simultaneously, significant improvement in performance can be gained. A collective communication system can be implemented for this segment since the reaction kinetics at each grid location is highly decoupled and only involves local flow quantities like temperature, pressure, density and internal energy.

Several aspects of KIVA influence the implementation and design of this parallel patch incorporated as part of this project. Grid adaptation is one of the key features of this software and hence the partitioning of the grid was required to be dynamic. Several features identifying solid and fluid domains needed to be resolved before implementing the parallel call as the CHEMKIN solver was only applied in the fluid domain. KIVA calculations have the additional complication that grid points become deactivated and activated during the course of a calculation but this feature was not an issue for our application since by the time the CHEMKIN routine was called the grid system was fixed. But this fixed grid changes in size for each time advance and hence the partitioning needed to be dynamic. Several checks on various conditions like crank angles, rpm, hydrodynamic pressure were continuously performed and this needed to be actively monitored during the iteration process. All the flow parameters were 1-D Vectors which made it extremely convenient for the parallel implementation as domain decomposition became trivial. Since the code was unstructured the following (Figure 1) grid notation was used for all the relevant flow parameters;

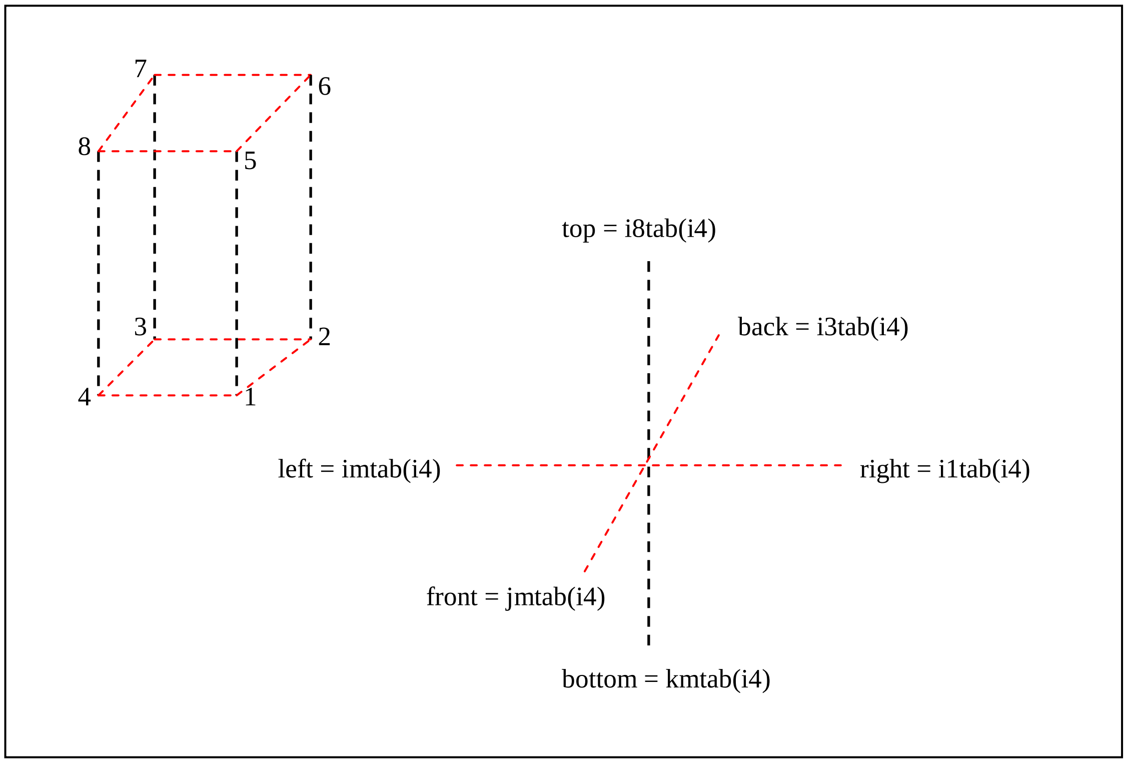

Figure 1: KIVA-3V grid notations; i4 refers both to vertex #4 and the cell shown.

3. Parallel Algorithm Design and Implementation

3.1 Master-Slave set-up

The parallel patch developed during the course of the project was based on a "modified" master-slave setup. This was a convenient model since parallelization was only focused on one segment of the code which was identified to be essential in improving performance. The flow solver was decoupled from the CHEMKIN segments thus breaking KIVA into three segments. The flow solvers were implemented in segments one and three and the chemistry was solved in segment two. The master node was responsible for dividing up the workload for individual processors. This was done using a collective communication operation as illustrated in the following sections. All the processors including the master processor was involved in computing the finite rate chemistry from CHEMKIN (This is in variation with the classical master-slave setup where the master node is only responsible in dividing the work among the slave nodes and gathering the results from them.) Once the solution from this segment was obtained another collective communication routine was initiated to integrate the global vectors back into the master node to continue into the final segments of the flow solver. This "modified" master-slave approach can be justified by the fact that the CHEMKIN subroutine was completely independent of the flow solvers and the master node operations were trivial and hence could be employed in computing the solution instead of remaining idle. The figure below demonstrates the KIVA parallelization approach used by the patch where Rank 0 is considered the Master node and the Rank 1,2,3 … n-1, are considered the slave nodes.

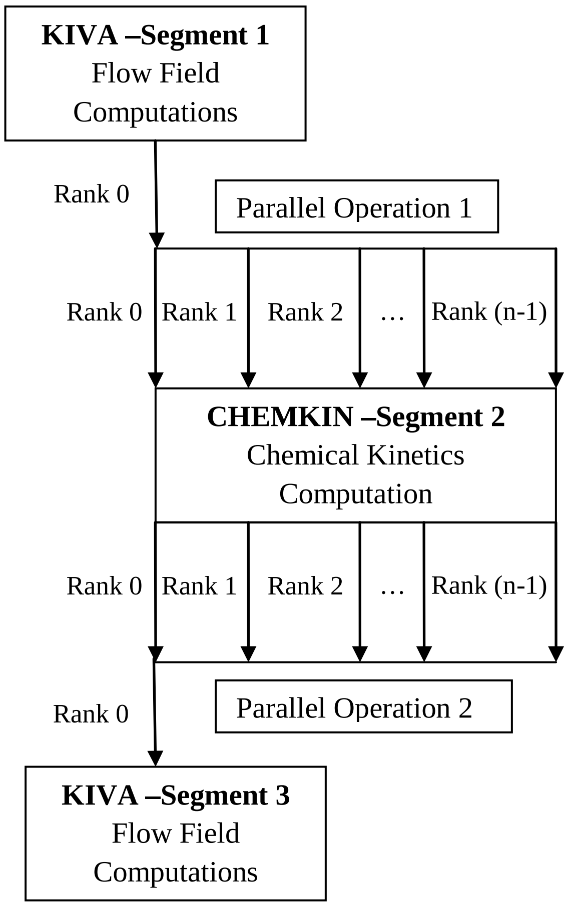

Figure 2: Parallelization patch implementation for KIVA using a master slave approach.

3.2 Domain Decomposition

Since CHEMKIN is essentially an ordinary differential equation solver, the domain decomposition is fairly easy. This is attributed to the fact that unlike partial differential equations (PDE) an ODE does not require any boundary conditions in the spatial domain at each local time-step. The only applicable boundary conditions are imposed in the time domain and since we are doing the parallel computation at each time step they are automatically imposed by the CHEMKIN solver. The flow characteristics or primitive variables associated with each grid point essential for the chemistry solve are density, temperature, pressure, internal energy and species density. These quantities are stored as vectors due to the unstructured set-up associated with the program (for the unstructured geometry notations refer Figure 1). Along with this the grid adaptation was also kept in mind before proceeding with the domain decomposition. In order to perform this operation the master processor with all the global variables was used to calculate the size of the domain. Depending on the number of processors the domain was split among all the *n* processors including the master node. Care was taken to ensure an almost even split was obtained even when the number of grid points were odd and also when the number of processors was odd in number. In order to perform the domain decomposition an MPI operation was performed before entering the CHEMKIN routine as shown in Figure 2. Details of this operation are as follows;

3.2.1 Parallel Operation 1

This operation involves splitting each of the various primitive variable vectors among all the processors including the master processor. The splitting was done using collective communication initiated by the master node, by utilizing the MPI subroutine call MPI_SCATTERV. The function of this operation is demonstrated in the following figure [5];

```
  MPI_SCATTERV(sendbuf, sendcount*, displ*, sendtype, recvbuf,
               recvcount,recvtype,root, comm, ierror)
```

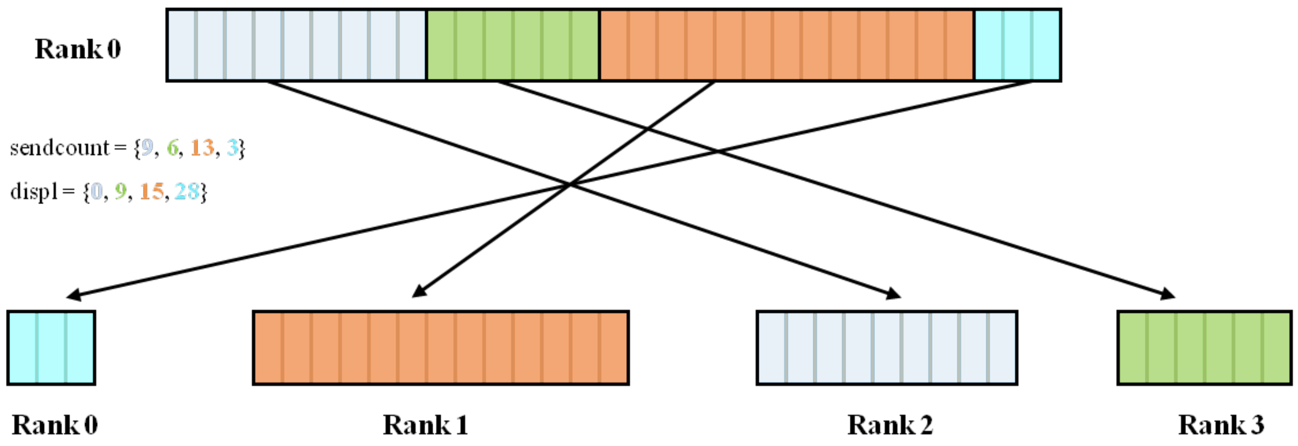

Figure 3: Domain decomposition using collective communication operator MPI_SCATTERV.

The sendcount and displ (displacement) vectors were dynamically computed by the master node and the recvbuffer was the local variable of each of the primitive variable being decomposed. Once this operation was performed each of the processors called the CHEMKIN solver and operations on the local primitives were carried out simultaneously by all the processors.

Since KIVA was written in FORTRAN 77 dynamic arrays were not used to compute the size of the local arrays. Instead a common block was defined where the local variables were initialized using certain arbitrary sizes keeping in mind the size of the domain. Care was taken to ensure that the size of these local arrays were large enough to process incoming and outgoing data.

Further, upon completion of the CHEMKIN (ODE solve) the local vectors were updated to new values and was reassembled or integrated back into the global array and this is indicated by the Parallel Operation 2 segment in Figure 2. Details of this operation are as follows;

3.2.2 Parallel Operation 2

This operation involves integrating each of the various local primitive variable vectors among all the processors including the master processor into a single global primitive variable vector. The gather operation was done using collective communication initiated by the master node, by utilizing the MPI subroutine call MPI_GATHERV. The function of this operation is exactly opposite to the MPI_SCATTERV operation demonstrated in Figure 3.

```
  MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf,
              recvcount*, displ*, recvtype, root, comm, ierror)
```

Since the chemistry operations were carried out on only the fluid elements of the primitive vectors care was taken to reassemble the vector in exactly the same order as before the split operation. This was important since the vectors contained both fluid and solid regions of the computational domain. The MPI_SCATTERV and MPI_GATHERV operations were extremely useful and convenient during these operations since the displacement of each of the segment of the global vector could be specified.

3.3 Processor Idling

Due to the “modified” master slave approach to the problem the pool of slave processors needed to be managed efficiently to be in sync with the master processor without carrying out any operations in several segments of the code. Hence during the serial implementation by the master node (KIVA routines) the slave nodes are allowed to idle and in order to accomplish this operation the MPI_BARRIER subroutine was called. This was essential since the slave processors need to be at the same part of the program when the master processor is ready to initiate the domain decomposition during each iteration. Since an entire parallelization of the KIVA code was not considered it was inevitable to have these processors idle since they were only employed in one segment of the code.

Finally, other useful collective communication subroutines like MPI_BCAST, MPI_REDUCE and others were used to perform several local calculations. A few write statements in KIVA were excluded from the algorithm due to time constraints and the validity of the code was tested by tracking the final solution. In order to analyze the performance of the code the MPI_WTIME subroutine was extensively employed at various portions of the code and this is discussed in the following sections.

In the following sections results from the performance analysis are presented and discussed with application to a standard 2-V engine.

4. Results and Discussion

4.1 Problem Formulation and Other Issues

In order to test the parallel performance the algorithm thus developed was validated using a standard 2-V engine geometry as shown in Figure 4.

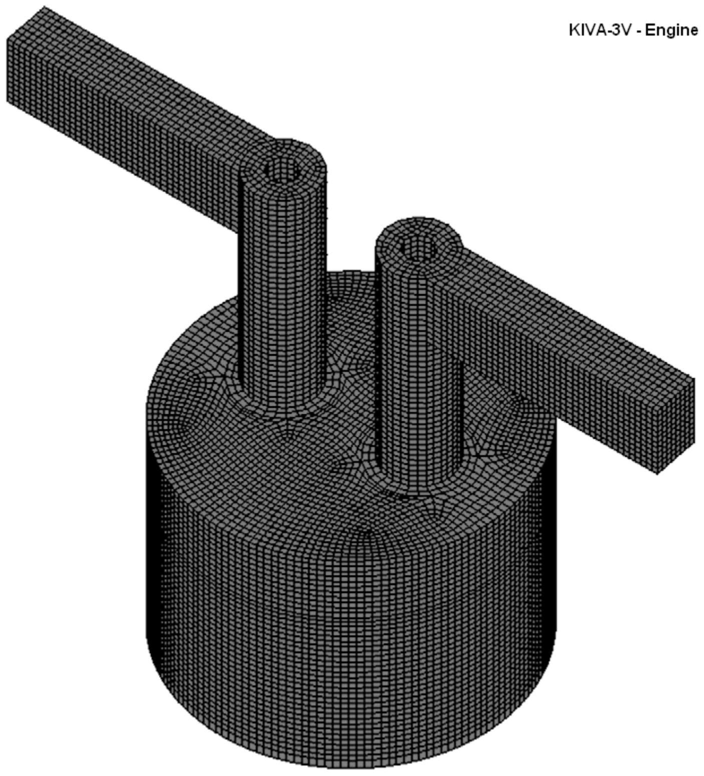

Figure 4: 2-V Engine geometry used for validation and testing [3].

In order to simulate the IC engine problem a crank angle variation from 180-360° was considered with standard conditions and a mesh size of 128,000 points. The CHEMKIN chemistry input files needed for simulation was extracted from the SD mechanism prescribed in CHEMKIN input files and similarly the thermodynamics data was correspondingly chosen in CHEMKIN. For lack of detailed chemistry mechanism only H_2, O_2, OH, N_2, H_2O, HO_2, H_2O_2, CO and CO_2 species were considered. The simulation accuracy can be improved by considering more accurate models for the reaction mechanisms and thermodynamic data. These considerations were limited by lack of input from other group members and time constraints and hence the confidence in the physics of the final solution is not completely validated. This factor was ignored since the focus of this project was on the parallel performance.

In order to evaluate the parallel performance two issues needed to be addressed; firstly, the overall performance of the algorithm can be evaluated but this might not be completely relevant to our application since the parallelization considered is localized to the CHEMKIN solve and the rest of the of the code is done in serial by the master processor, second, a performance evaluation of only the CHEMKIN solve can be evaluated for performance and its scalability can be studied since this involves parallelization of this entire segment. In order to address this issue the following arguments of Gustafson's Law for speed-up was considered.

4.2 Gustafson's Law

Gustafson's Law is closely related to Amdahl's law and is formulated as;

$$S(P) = P - \alpha(P-1)$$

where P is the number of processors, S is the speed-up and α the non-parallelizable part of the process. Hence, in our problem the KIVA section of the code was assumed to be non-parallelizable and the CHEMKIN section was considered the parallelized portion [6]. This above law can be suitably applied to study the performance of the parallel algorithm developed during the course of the project and can be used as a reference solution besides the ideal linear speed-up model. Gustafson's law is similar to Amdahl's law where the non-parallelizable section is considered the portions of the code that have been parallelized but may not be solved in parallel but this model can be extrapolated to the current problem since several segments of the code are executed in serial. In order to compute the constant alpha the timer in the program was setup to calculate the times from each segment of the code and the final fraction of the serial segments was calculated.

4.2 Speed-up and Efficiency Analysis

In order to analyze the performance of the algorithm the speed-up and efficiency can be computed. The speed-up was defined by the following formula;

$$S(P) = \frac{Execution\ time\ on\ a\ single\ process}{Execution\ time\ on\ 'p'\ processors} = \frac{T_1}{T_P}$$

The efficiency was defined as;

$$E_P = \frac{S_P}{P}$$

4.5 Parallel Performance

The problem set-up in KIVA was run on CAC-nyx cluster with the nodes that have dual-dual core Opteron CPU's at the University of Michigan. Summarized below are the timing, speed-up and efficiency obtained from the 2-V engine runs;

Number of Processors	Time (hrs)	Speed-up	Efficiency (%)
1	6.05	1	100
2	4.46	1.356502	67.82511211
4	2.95	2.050847	51.27118644
8	1.77	3.418079	42.7259887
16	1.05	5.761905	36.01190476

In order to have a reference solution to compare the above parallel performance the Gustafson’s law was considered where fractions of the code spent in doing the serial KIVA routines was calculated and the fraction of time spent in the serial segments of the code was found to be α=0.66. From this analysis it was concluded that only 33% of the code is parallelized and its performance was evaluated.

1-processor	**Time (hr)**	**Percent (%)**
KIVA - Segment 1	1.68	27.768595
CHEMKIN	2.01	33.2231405
KIVA - Segment 2	2.36	39.0082645
Total	**6.05**	

The following performance analysis plot was obtained;

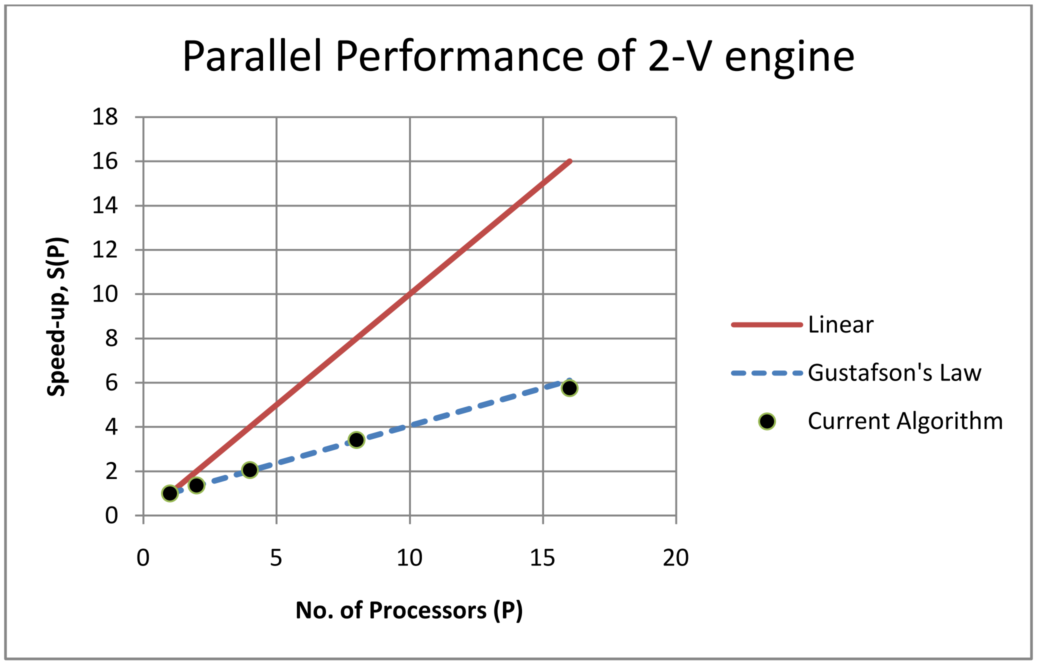

Figure 5: Parallel Performance of 2-valve engine problem.

From the above plot it can be concluded that an improvement in performance was achieved in computing the reaction kinetics of the KIVA-3V engine code but as evident in the above plot variations from the Gustafson’s law are observed with 16 processors. Due to time constraints a higher processor run couldn’t be considered (p = 32, 64). Hence the strong scalability of the above algorithm may not completely satisfactory and improvements in the algorithm may need to be considered. An overall improvement in performance is definitely achieved which was the goal of the project. This also gives an insight into the gains of parallelizing the KIVA algorithm.

4.5 Validation Case

In order to validate the code the following plots for total mass (fuel) plotted along four cut planes for couple of runs with 1 and 8 processors are presented below;

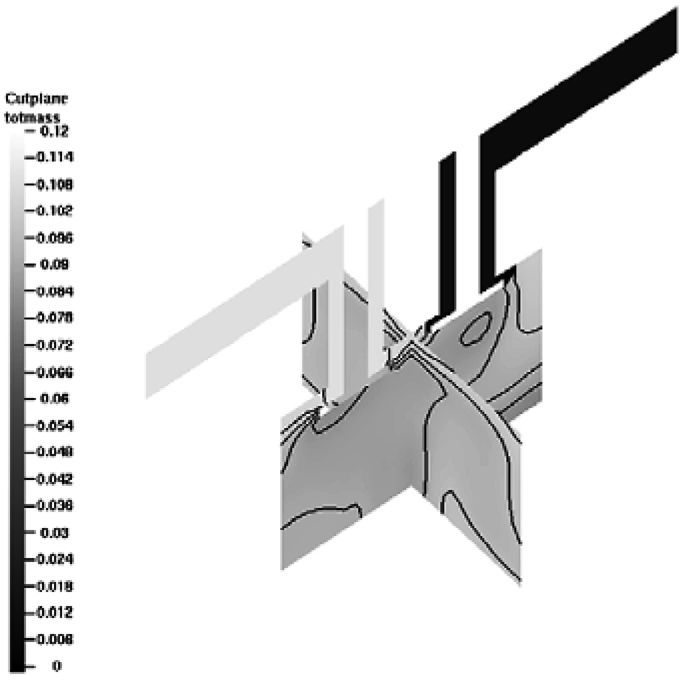

Figure 6: Cross-sectional fuel contours with 1 processor (Validation Case).

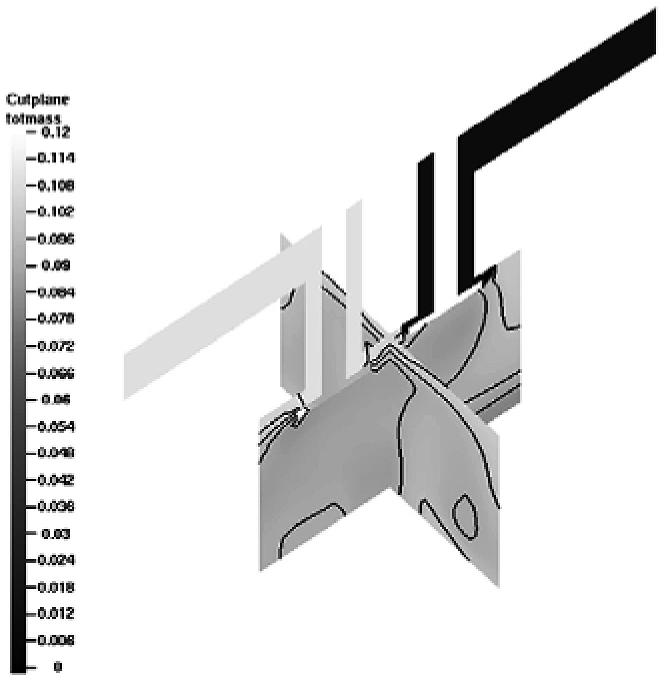

Figure7: Cross-sectional fuel contours with 8 processors (Validation Case).

From the above plots, the fuel distribution compares reasonably well in both simulations despite some differences in the fuel contours. This can be attributed to the fact that some of the chemistry mechanism used during the runs was adjusted since detailed knowledge on these settings was not available. This was due to the lack of input from other members in the group. Hence the physics of the solution is not completely validated.

5. Conclusion

From the above analysis it can be concluded that the results obtained were satisfactory and the parallelization applied to the CHEMKIN call segment seems to give a reasonable performance. A speed-up in comparison to the Gustafson's Law was achieved and in order to further improve the performance i.e. to get a linear speed-up, the parallelization of the entire code needs to be considered. The only constraint in parallelizing KIVA is that the code is written in FORTRAN 77 and is extremely tedious and time consuming to parallelize the entire algorithm. This was also experienced during the course of this project.

5. References

[1] Yasar O, Zacharia T. Distributed implementation of KIVA-3 on the Intel Paragon in parallel computational fluid dynamics: implementations and results using parallel computers. New York: Elsevier; 1995. p. 81–8.

[2] D' Ambra P, Filipone S, Nobile P. The Use of Parallel Numerical Library in Industrial Simulation: The case study of the Design of a Two Stroke Engin. New York: Elsevier;1996.

[3] Torres D J, Li Y H, Kong S-C. Partitioning strategies for parallel KIVA-4 engine simulation. Los Alamos: Computers & Fluids; 2009, p.301-309

[4] Bella G, Bozza F, De Maio A, Del Citto F, Filippone S. An enhanced parallel version of KIVA-3V coupled with a 1D CFD code, and its use in general purpose engine applications. Lect Notes Comput Sci 2006;4208:11–20.

[5] Gropp W, Huss-Lederman S, Snir M. MPI-The Complete Reference; Sci. and Engg. Comp. Series, Vol. 2; New York;1998

[6] Gustafson J. Reevaluating Amdahl's Law. Communications of the ACM 31(5); 1988; p. 532-533

[7] Cyberinfrastructure Tutor in Parallel Computing. Introduction to MPI.

6. Appendix

Parallel Patch used to parallelize KIVA-3V. Presented below are the subroutines that were created and modified as part of this project.

- Additions to the common block comkiva.i

```
!AS PAR Mod - 12/01/2009: added gandharv
      integer nproc,irank,iroot,ierr
      real wtime
      common /gandharv/ nproc,irank,iroot,ierr,wtime

      parameter (nv_=10000)
      real ro_,p_,spd_,temp_,vol_,sie_
      integer idreg_
      common/parallel/ro_(nv_),p_(nv_),spd_(nv_,lnsp),
     1 temp_(nv_),vol_(nv_),sie_(nv_),idreg_(nv_)

      integer i4_,ifirst_,ncells_,flag,i4proc
      common/indexing/ i4_,ifirst_,ncells_,flag,i4proc

      real t0_seg1,t1_seg1,t0_seg2,t1_seg2,tf_seg1, tf_seg2
      common/timing/ t0_seg1,t1_seg1,t0_seg2,t1_seg2,tf_seg1, tf_seg2
```

- Driver parallel_kiva.f

```
*deck kiva
      program kiva
      include 'comkiva.i'
!ACB 10-11-04 Added include statement below
      include 'comsnap.i'
      external fuelib
!ACB 10-11-04 Added logical statement below
      logical eeexxx
!GK 12-01-09 Parallelization
      include 'mpif.h'

c <><><><> Initializing the Parallel Environment<><><><><><><><><><><><>
      call parallel_init
c <><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
c
c +++
c +++ if lospeed=1.0, rinput sets perrmx = 1.0e-2 and epsp = 1.0e-5
c +++ if pmplict = 1.0 or 2.0, rinput sets maxpit = 50

c Segment KIVA-1
      t0_seg1 = MPI_WTIME()
c AS PAR Mod - 12/01/2009: added gandharv
 111  if (irank.eq.iroot) then
         perrmx  = 0.10
         nbigtst = 50
c AS RCM Mod - 10/02/2009: added askogmv
         askogmv = 1
         call qstart
         call begin
         if(irest.gt.0) call taperd
         call rinput
         if(irest.eq.0) call setup
         if(irest.gt.0) call second (tbegin)
      elseif (irank.ne.iroot) then
         goto 999
  112 end if

  113 if (irank.eq.iroot) then
         if(irest.gt.0) go to 20
      elseif (irank.ne.iroot) then
         goto 999
  114 end if

  10  if (irank.eq.iroot) then
         call second (tbegin)
         if(nohydro.eq.1) go to 15
         call visc
      elseif (irank.ne.iroot) then
         goto 999
  110 end if

  15  if (irank.eq.iroot) then
         if(irecyc.eq.1) call timenrest
         call timstp
      elseif (irank.ne.iroot) then
         goto 999
  115 end if

  20  if (irank.eq.iroot) then
         call newcyc
c AS RCM Mod - 10/02/2009
c        if(rpm.gt.0.0) call piston
         call piston
         if(nohydro.eq.1) go to 55
         if(numinj.gt.0) then
           do 30 n=1,numinj
              if(t.ge.t1inj(n) .and. t.le.t2inj(n)) call inject(n)
   30         continue
              endif
           if(np.gt.0) call pmovtv
           if(np.gt.0 .and. breakup.eq.1.0) call break
           if(np.gt.0 .and. kolide.eq.1) call colide
           evapflg=0.0
           if(np.gt.0 .and. evapp.eq.1.0) call evap
           if(lwall.eq.+1) call lawall
           if(lwall.eq.+1 .and. np.gt.0) call wallfilm
           if(anc4o2.gt.0.0) call nodcpl
           if(isoot.eq.1) call soot
c-----------------------------------------------------------------------
!ACB 10-11-04 output cloud information at first 5 timesteps
           inquire(file='cells_tcute5.out',exist=eeexxx)
           if (eeexxx) goto 601
              inquire(file='cells_tcute0.out',exist=eeexxx)
           if (eeexxx) then
              open(97, file='cells_tcute0.out')
              read(97,*) ncyctcute
              close(97)
              if (ncyc.eq.(ncyctcute+1)) then
               open(unit=97, file='cells_tcute1.out', status='unknown')
               iout=1
               goto 599
              endif
              if (ncyc.eq.(ncyctcute+2)) then
               open(unit=97, file='cells_tcute2.out', status='unknown')
               iout=2
               goto 599
              endif
              if (ncyc.eq.(ncyctcute+3)) then
               open(unit=97, file='cells_tcute3.out', status='unknown')
               iout=3
               goto 599
              endif
              if (ncyc.eq.(ncyctcute+4)) then
               open(unit=97, file='cells_tcute4.out', status='unknown')
               iout=4
               goto 599
              endif
              if (ncyc.eq.(ncyctcute+5)) then
               open(unit=97, file='cells_tcute5.out', status='unknown')
               iout=5
               goto 599
              endif
              goto 601
 599          write(97,*) ncyc,
c     *            '  cell_mass    cell_temp    cell_sphi    cell_bphi  '
              do i4=ifirst,ncells
               if(idreg(i4).eq.1) then
                 cccmass=ro(i4)*vol(i4)
                 sssphi=smallphi(spd(i4,1:nsp))
                 bbbphi=bigphi(spd(i4,1:nsp))
                 write(97,600) iout, cccmass,temp(i4),sssphi,bbbphi
 600             format(x,i2,4(1x,e12.5))
               endif
             end do
             close(97)
           end if
      elseif (irank.ne.iroot) then
          goto 999
 120  endif
 601  continue

      t1_seg1 = MPI_WTIME()
c CHEMKIN Segment - 2
c----------------------------------------------------------------------
 999  call MPI_BARRIER(MPI_COMM_WORLD,ierr)
c      print*, "Scatter Operation Begun...",irank
      flag = 1
      call partition
      call chem_ck
c      print*, "Gather Operation Begun...",irank
      flag = 2
      call partition
c----------------------------------------------------------------------
```

```
c      print*, 'irank=',irank, iroot
c*******************PRINTING AND CHECKING PAR****************************
c       if (irank.eq.1) then
c         print*, 'irank=,',irank
c         print*, 'ctempmax=',ctempmax
c         print*, 'dt=',dt
c         print*, 'tcute=',tcute
c         write(11,*) 'i4_=',i4proc,'ifirst_=',ifirst_,'ncells_=',ncells_
c         write(11,*) 'ro_=',ro_(ifirst_:ncells_)
c       elseif (irank.eq.2) then
c         print*, 'irank=,',irank
c         print*, 'ctempmax=',ctempmax
c         print*, 'dt=',dt
c         print*, 'tcute=',tcute
c         write(12,*) 'i4_=',i4proc,'ifirst_=',ifirst_,'ncells_=',ncells_
c         write(12,*) 'ro_=',ro_(ifirst_:ncells_)
c       else
c         print*, 'irank=,',irank
c         print*, 'ctempmax=',ctempmax
c         print*, 'dt=',dt
c         print*, 'tcute=',tcute
c         write(10,*) 'i4_=',i4proc,'ifirst_=',ifirst_,'ncells_=',ncells_
c         write(10,*) 'ro=',ro(ifirst:ncells)
c       end if
c      stop
c************************************************************************


c KIVA Segment - 3
      t0_seg2 = MPI_WTIME()
 1113 if (irank.eq.iroot) then
         if(gx.ne.0.0 .or. gy.ne.0.0 .or. gz.ne.0.0) call gravity
         if(np.gt.0) call pmom
         if(np.gt.0 .or. evapflg.eq.1.0) call pcoupl
         if(irecyc.eq.1) go to 15
         if(np.gt.0) call repack
         call ysolve
         if(irecyc.eq.1) go to 15
         call exdif
         call pinit
         if(phidmx.gt.0.0) call pgrad (one)
         ibigit=0
      elseif (irank.ne.iroot) then
         goto 999
 1114 end if

 40   if (irank.eq.iroot) then
         ibigit=ibigit+1
         call vsolve
         if(irecyc.eq.1) go to 15
         call tsolve
         if(irecyc.eq.1) go to 15
         call psolve
         if(perr.lt.perrmx .and. numpit.lt.maxpit) go to 50
         if(ibigit.eq.nbigtst) then
           irecyc=1
           go to 15
         endif
         if(phidmx.gt.0.0) call pgrad (three)
         go to 40
      elseif (irank.ne.iroot) then
         go to 999
 140  end if

 50   if(irank.eq.iroot) then
        call pgrad (one)
        call phaseb
        if(tkesw.eq.1.0) call kesolv
        if(irecyc.eq.1) go to 15
        if(np.gt.0) call paccel
      elseif (irank.ne.iroot) then
        goto 999
 150  end if

 55   if(irank.eq.iroot) then
        call rezone
        if(nohydro.eq.1) call convex
        if(irez.gt.0) call volume
        if(nohydro.eq.1) go to 65
        if(irez.gt.0) call aproj
        do 60 nsub=1,nfluxs
        call ccflux(nsub)
        call momflx(nsub)
 60     continue
      elseif (irank.ne.iroot) then
        goto 999
 155  end if

 65   if (irank.eq.iroot) then
         if(rpm*snapper.gt.0.0) then
c-----------------------------------------------------------------------
c           if(nvfaceb(0).gt.0) call snapb
!ACB 05-05-04 commented above added below for snapping with fake bowl
            if(nvfaceb(0).gt.0) call snapb(0,180.0)
c-----------------------------------------------------------------------
            if(twopiston.eq.1.0) then
              if(nvfacet(1).gt.0) call snapt
            else
              if(nvfaceb(2).gt.0) call snapvtop
              if(nvfacet(1).gt.0) call snapvfce
            endif
         endif
         if(nohydro.eq.0) call state
         if(irecyc.eq.1) go to 15
         call second (tcycle)
         timetotal=timetotal + (tcycle-tbegin)
         if(nohydro.eq.0) go to 10
         call second (tbegin)
         go to 20
      elseif (irank.ne.iroot) then
         go to 999
 165  end if
      t1_seg2 = MPI_WTIME()
```

```
c <><><><><>Finalizing the Parallel Environment <><><><><><><><><><><><>
      call parallel_final
c <><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
      tf_seg1 = t1_seg1-t0_seg1
      tf_seg2 = t1_seg2-t0_seg1

      print*, 'Time spent in KIVA-segment-1=', tf_seg1
      print*, 'Time spent in KIVA-segment-2=', tf_seg2

      end
```

- Initialize, Finalize and wall time in parallel

```
c Some parallel subroutines
c 1) Initialize Environment
c 2) Finalize Environment

      subroutine parallel_init

            include 'comkiva.i'
            include 'mpif.h'

            call MPI_INIT(ierr)
            call MPI_COMM_RANK(MPI_COMM_WORLD,irank,ierr)
            call MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,ierr)
            iroot = 0

            return
      end

      subroutine parallel_final
            include 'comkiva.i'
            include 'mpif.h'

            call MPI_FINALIZE(ierr)

            return
      end
```

- Partitioning and Collective Communication routines, partition.f

```
c Root Processor does partitioning and distributes global arrays
c to Slaves which are idle

      subroutine partition

         include 'comkiva.i'
         include 'mpif.h'

         integer G1,G2,G3,i
         integer i1_(nproc),nC_(nproc),i4par(nproc), iproc(nproc)

         if (irank .eq. iroot) then
c             write(123,*) "sie=",sie(ifirst:ncells)
             G3=(ncells-ifirst)+1
             do i=1,nproc
                iproc(i) = i-1
             end do
             G1=G3/nproc
             G2=mod(G3,nproc)

             do i=1,nproc
                if (iproc(i).lt.G2) then
                   i4par(i)  = G1+1
                   i1_(i) = ifirst + iproc(i)*(G1+1)
                else
                   i4par(i)  = G1
                   i1_(i) = ifirst + G2*(G1+1) + (iproc(i)-G2)*G1
                endif
                   nC_(i) = i1_(i) + i4par(i) - 1
             end do

         end if
         call MPI_BARRIER(MPI_COMM_WORLD,ierr)
         if (flag.eq.1) then
            call scatter(i1_,nc_,i4par)
         elseif (flag.eq.2) then
            call gather(i1_,i4par)
         end if
         return
      end

      subroutine scatter(s1p,s2p,scp)
         include 'mpif.h'
         include 'comkiva.i'

         integer s1p(nproc),s2p(nproc),scp(nproc)
c Communication to Slave Processors
c Scalars
         call MPI_SCATTER (scp,1,MPI_INTEGER,i4proc,1,
     &        MPI_INTEGER,iroot,MPI_COMM_WORLD,ierr)
         call MPI_SCATTER (s1p,1,MPI_INTEGER,ifirst_,1,
     &        MPI_INTEGER,iroot,MPI_COMM_WORLD,ierr)
         call MPI_SCATTER (s2p,1,MPI_INTEGER,ncells_,1,
     &        MPI_INTEGER,iroot,MPI_COMM_WORLD,ierr)
         call MPI_BCAST (dt,1,MPI_DOUBLE_PRECISION,iroot,
     &        MPI_COMM_WORLD,ierr)
c         call MPI_BCAST (tcute,1,MPI_DOUBLE_PRECISION,iroot,
c     &        MPI_COMM_WORLD,ierr)
c Vectors
         call MPI_SCATTERV (ro,scp,s1p-1, MPI_DOUBLE_PRECISION,
     &        ro_(ifirst_:ncells_),scp,MPI_DOUBLE_PRECISION,
     &        iroot,MPI_COMM_WORLD,ierr)
         call MPI_SCATTERV (p,scp,s1p-1, MPI_DOUBLE_PRECISION,
     &        p_(ifirst_:ncells_),scp,MPI_DOUBLE_PRECISION,
     &        iroot,MPI_COMM_WORLD,ierr)
         call MPI_SCATTERV (vol,scp,s1p-1,MPI_DOUBLE_PRECISION,
     &        vol_(ifirst_:ncells_),scp,MPI_DOUBLE_PRECISION,iroot,
     &        MPI_COMM_WORLD,ierr)
         call MPI_SCATTERV (temp,scp,s1p-1,MPI_DOUBLE_PRECISION,
     &        temp_(ifirst_:ncells_),scp, MPI_DOUBLE_PRECISION,iroot,
     &        MPI_COMM_WORLD, ierr)
         call MPI_SCATTERV (spd,scp,s1p-1,MPI_DOUBLE_PRECISION,
     &        spd_(ifirst_:ncells_,:),scp,MPI_DOUBLE_PRECISION,
     &        iroot,MPI_COMM_WORLD,ierr)
         call MPI_SCATTERV (sie,scp,s1p-1,MPI_DOUBLE_PRECISION,
     &        sie_(ifirst_:ncells_),scp,MPI_DOUBLE_PRECISION,
     &        iroot,MPI_COMM_WORLD,ierr)
         call MPI_SCATTERV (idreg,scp,s1p-1,MPI_INTEGER,
     &        idreg_(ifirst_:ncells_),scp,MPI_INTEGER,
     &        iroot,MPI_COMM_WORLD,ierr)
        return
      end

      subroutine gather(s1_,scp_)
         include 'mpif.h'
         include 'comkiva.i'

         integer s1_(nproc),scp_(nproc)

c Communication to Slave Processors
c Vectors
         call MPI_GATHERV (ro_(ifirst_:ncells_),i4proc,
     &        MPI_DOUBLE_PRECISION,ro,scp_,s1_-1,
     &        MPI_DOUBLE_PRECISION,iroot,MPI_COMM_WORLD,ierr)
         call MPI_GATHERV (p_(ifirst_:ncells_),i4proc,
     &        MPI_DOUBLE_PRECISION,p,scp_,s1_-1,
     &        MPI_DOUBLE_PRECISION,iroot,MPI_COMM_WORLD,ierr)
         call MPI_GATHERV (vol_(ifirst_:ncells),i4proc,
     &        MPI_DOUBLE_PRECISION,vol,scp_,s1_-1,
     &        MPI_DOUBLE_PRECISION,iroot,MPI_COMM_WORLD,ierr)
         call MPI_GATHERV (temp_(ifirst_:ncells_),i4proc,
     &        MPI_DOUBLE_PRECISION,temp,scp_,s1_-1,
     &        MPI_DOUBLE_PRECISION,iroot,MPI_COMM_WORLD,ierr)
         call MPI_GATHERV (spd_(ifirst_:ncells_,:),i4proc,
     &        MPI_DOUBLE_PRECISION,spd,scp_,s1_-1,
     &        MPI_DOUBLE_PRECISION,iroot,MPI_COMM_WORLD,ierr)
         call MPI_GATHERV (sie_(ifirst_:ncells_),i4proc,
     &        MPI_DOUBLE_PRECISION,sie,scp_,s1_-1,
     &        MPI_DOUBLE_PRECISION,iroot,MPI_COMM_WORLD,ierr)
         call MPI_GATHERV (idreg_(ifirst_:ncells_),i4proc,
     &        MPI_INTEGER,idreg,scp_,s1_-1,
     &        MPI_INTEGER,iroot,MPI_COMM_WORLD,ierr)
        return
      end
```

- Parallelized CHEMKIN call routine parallel_chem_ck.f

```
      subroutine chem_ck
c ========================================================================
!      calculates the change in species densities and internal energy
!      due to kinetic chemical reactions
!
!      chem_ck is based on subroutine chem.  Instead of using the
!      standard kiva kinetic chemical reactions, ck_solv is called and
!      species densities and internal energies are updated.
!
!      chem is called by:  kiva
```

```
!
!      chem calls the following subroutines and functions:
!                            ck_solv, ck_init, exitk
c ======================================================================
      include 'comkiva.i'
      include 'comchem.i'
      include 'mpif.h'

      dimension cmass(nv_), cphi(nv_)

      dimension tmp(lnsp), tmp2(lnsp)
      dimension zyck(lnsp), ycell(lnsp)

c**********************************************************************
c Initialize auxiliary variables
      ctempmax_ = 0.    !max cell temperature in cylinder
      ctempmax  = 0.
      tmass    = 0.     !total mass in cylinder
      cmass    = 0.     !cell mass
      sphi     = 0.     !cell equivalence ratio (small PHI)
      bphi     = 0.     !cell equivalence ratio (big PHI)
      cphi     = 0.     !cell PHI to be used for sorting (sphi or bphi)
      ncr1     = 0      !number of cells in region 1
      do isp=1,nsp
        tmp(isp) = 0.0  !species masses in cylinder (start)
        tmp2(isp)= 0.0  !species masses in cylinder (end)
      enddo

      temp0      = 300.     !reference temperature for enthalpy

c      dcl = 5.              !Cloud output interval
c     print*, "tcutflag=", tcutflag
c Keep only cells with idreg=1

      do i4_=ifirst_,ncells_
        cmass(i4_)=ro_(i4_)*vol_(i4_)
        sphi(i4_)=smallphi(spd_(i4_,1:nsp)) ! Function call ck_fun.f
        bphi(i4_)=bigphi(spd_(i4_,1:nsp))   ! Function call ck_fun.f
        ycell(1:nsp_) = spd_(i4_,1:nsp)/ro_(i4_)
        if(idreg_(i4_).eq.1) then
          tmass=tmass + cmass(i4_)
          ctempmax_=max(ctempmax_,temp_(i4_))
          ncr1=ncr1+1              !number of cells in Region 1
          do isp=1,nsp
            tmp(isp)=tmp(isp)+spd_(i4_,isp)*vol_(i4_)
          enddo
        endif
      enddo

c Need to find ctempmax in the entire domain
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      call MPI_REDUCE(ctempmax_,ctempmax,1,MPI_DOUBLE_PRECISION,
     &                MPI_MAX,iroot,MPI_COMM_WORLD,ierr)
      call MPI_BCAST(ctempmax,1,MPI_DOUBLE_PRECISION,iroot,
     &                MPI_COMM_WORLD,ierr)
c Broadcast tcutflag to all procs
      call MPI_BCAST (tcutflag,1,MPI_LOGICAL,iroot,
     &        MPI_COMM_WORLD,ierr)
c Broadcast tcute to all procs
      call MPI_BCAST (tcute,1,MPI_DOUBLE_PRECISION,iroot,
     &        MPI_COMM_WORLD,ierr)

c Check if tcute has been reached; if yes, continue
      if (.not.tcutflag) then
        if (ctempmax.lt.tcute) then
          tmaxprev=ctempmax
          goto 2000
        endif
        if (((tcute-tmaxprev)/tcute).gt.0.01) then
          print*,'passed tcute, but tmaxprev =',tmaxprev
          tmaxprev=ctempmax
          goto 2000
        endif
        print*, 'passed tcute', tmaxprev, ctempmax
        tcutflag = .true.
      endif
c!*********************************************************************

c**********************************************************************
c Compute average P, T, composition (zyck) and PHI for each zone
c Compute number of c/h/o atoms in the zone

      ickcall = 0
      do 200 i4_=ifirst_,ncells_
        if(idreg_(i4_).eq.1) then
          zyck(1:nsp) = spd_(i4_,1:nsp)*vol_(i4_)
          zmasschk = ro_(i4_)*vol_(i4_)
          zavgt = temp_(i4_)
          zavgp = p_(i4_)
          zvol = vol_(i4_)

c**********************************************************************
c Call ck_solv with average values for T/PHI zone
          dechem=0.0

          if (zavgt.lt.tcute) goto 200
          call ck_solv(dt,zavgt,zavgp,zyck(1:nsp),dechem)
          ickcall = ickcall + 1

c**********************************************************************
c Assign new spd and dechemi4 to cell

          spd_(i4_,1:nsp) = zyck(1:nsp)/vol_(i4_)
          ycell(1:nsp) = zyck(1:nsp)/zmasschk
          sie_(i4_) = sie_(i4_) + dechem
c
          wchem = wchem + dechem*zmasschk*facsec

c SANITY CHECK 8
          if(sie_(i4_).lt.0) then
            print*,i4_,sie_(i4_),dechem
            write(*,*) 'chem_ck SANITY CHECK 8 - termination!'
            call exitk(55)
          endif
```

```
c END CHECK 8

        endif
 200  continue

 2000  return
      end
```