



Porting and optimizing HYDRO to new platforms and programming paradigms – lessons learnt

Pierre-François Lavallée^a, Guillaume Colin de Verdière^b, Philippe Wautelet^a,
Dimitri Lecas^a, Jean-Michel Dupays^a

^aIDRIS/CNRS, Campus universitaire d'Orsay, rue John Von Neumann, Bâtiment 506, F-91403 Orsay, France

^bCEA, Centre DAM Ile-de-France, Bruyères-le-Châtel, F-91227 Arpajon, France

Abstract

The purpose of low-level benchmarks is to measure certain important characteristics of the target computer system such as arithmetic and communication rates and overheads. They are synthetic in the sense that each is designed to measure a particular architectural feature of the computer. In contrast to higher-level kernel and application benchmarks, they solve no real problem and they don't exhibit properties of real production codes which scientific computer developers experiment daily with.

On the other hand real application codes can be very complex and can use multiple specific algorithms. It can be very difficult or costly to port the code to a specific processor or to a new architecture.

Since HYDRO has been extracted from a real code (RAMSES [1]), it occurred to us that it will be a good candidate for benchmarking purposes. HYDRO includes classical algorithms we can find in many applications codes for Tier-0 systems.

It has been written in several versions including Fortran and C in order to experiment many new ways of parallelism and to adapt it easily to new architectures that are emerging.

In this paper, we described the different versions of HYDRO we have developed using classical or new parallel programming techniques or paradigms. We also synthesized the lessons that could be learned from this work, the difficulties that we have encountered in porting the application, the ease of use and the maturity of the new parallel programming paradigms and the significant improvements in terms of performance that could be obtained.

Table of contents

1	Introduction	3
2	Algorithms	4
3	The Fortran Branch.....	7
3.1	Sequential version	7
3.2	OpenMP Fine-Grain version.....	10
3.3	OpenMP Coarse-Grain version.....	13
3.4	MPI version.....	17
3.5	Performance of OpenMP Fine-Grain, OpenMP Coarse-Grain and MPI version	19
3.6	Hybrid MPI/OpenMP version.....	22
4	The C branch	27
4.1	C89.....	27
4.2	CUDA 1D	31
4.3	OpenCL.....	36
4.4	HMPP.....	37
4.5	C99.....	40
4.6	C99 MPI 2D	41
4.7	CUDA MPI 2D	42
4.8	UPC version	43
4.9	The VTK output.....	53
4.10	Test cases.....	54
5	Conclusions	55
6	Acknowledgements	55
7	References	55

1 Introduction

RAMSES [1] was developed in astrophysics' division in CEA Saclay (France) by Romain Teyssier (CEA) to study large scale structure and galaxy formation. It is written in FORTRAN with extensive use of the MPI library. It is well known in the Computational Fluid Dynamics community and has proven to be scalable to tens of thousands of processes.

HYDRO is a simplified version of RAMSES:

- The space domain is a rectangular two-dimensional splitting with a regular cartesian mesh (there is no Adaptive Mesh Refinement),
- HYDRO solves compressible Euler equations of hydrodynamics,
- HYDRO is based on a finite volume numerical method using a second order Godunov scheme [2] for Euler equations,
- a Riemann solver [3] computes numerical flux at the interface of two neighbouring computational cells.
- HYDRO code has about 1500 lines.

HYDRO is neither a small kernel nor a big software. It uses common algorithms representative of the ones found in HPC. Hence, it seems to be a good candidate to fulfil the goals of this project:

- study and assess classical parallelization technics and more advanced programming frameworks related to accelerators or ARM based machines,
- compare the performance and parallel scalability of a wide variety of architectures.

In order to cope with the wide varieties of approaches that we want to assess, two main branches of the code have been developed:

1. The initial FORTRAN branch including OpenMP [5], MPI [4], hybrid MPI/OpenMP approaches [7],
2. A C branch to mainly facilitate the porting of the application on GPU platforms including CUDA, OpenCL, HMPP programming frameworks and novel HPC languages including UPC [9,10].

All versions have been developed trying to keep up the following rules:

- the algorithms have not been modified,
- as much as we can, the code structure has been kept,
- the results have been validated by comparison with the sequential version.

2 Algorithms

The two-dimensional Euler equations of hydrodynamics for an ideal gas expressing the conservation of mass, momentum and energy can be written as:

$$\partial_t U + \partial_x F(U) + \partial_y G(U) = 0$$

where:

$U = (\rho, \rho u, \rho v, E)$ is the vector of conservative variables,

ρ, u, v, E are respectively the density, the x- and y-velocities and the total energy,

p is the pressure with $\gamma = 1.4$ (for H_2 at temperature 100°C) often used in astrophysics simulation,

$$p = (\gamma - 1) \left[E - \frac{\rho}{2} (u^2 + v^2) \right]$$

F and G are the flux vectors.

The two-dimensional Euler equations in conservative form are discretized in the finite volume framework as follows:

$$U_{i,j}^{n+1} = U_{i,j}^n + \frac{\Delta t}{\Delta x} (F_{i+1/2,j}^{n+1/2} - F_{i-1/2,j}^{n+1/2}) + \frac{\Delta t}{\Delta y} (G_{i,j+1/2}^{n+1/2} - G_{i,j-1/2}^{n+1/2}) \quad (1)$$

where the flux functions are time and space averaged. In a cartesian grid, the elementary grid cell is simply a square which the centre is $(x=i, y=j)$ of sizes $\Delta x, \Delta y$.

In Table 1 we summarize the Godunov scheme with splitting direction technique.

```

initialize uold(:, :, :)      // variable uold to store  $U_{i,j}^0$ 

initialize  $n_{step} = 0$       // discrete time variable

while  $t < t_{end}$  do

    if  $n_{step} \% n_{output} == 0$  then

        output();           // dump fluid variables
        Dt = cmpdt();        // compute time step

        if  $n_{step} \% 2 == 0$  then

            Godunov(X, dt); Godunov(Y, dt); // update uold()

        else

            Godunov(Y, dt); Godunov(X, dt); // at t+dt

        end if

    end while

generate timing report

```

Table 1. Directional splitting Godunov scheme algorithm

In Table 2 we describe the pseudo-code of the routine implementing the equation (1) to update fluid cells $uold(:, :, :)$. At each time step, the Godunov routine is called twice, once in each direction (see Figure 1-2).

```

make_boundary();           // Apply boundary conditions to uold(:, :, :)

// Update on the first direction (idim=1), column by column
for j=1,ny do
    • u(:,j)=uold(:,j,:) // copy of one column of uold in the working buffer u
    • Compute primitive variables  $q(:,j) = (\rho, u, v, p)$ 
    • Solve Riemann problem at current cell interfaces, i.e compute Godunov state
    • Compute incoming fluxes  $F_{i+1/2,j}^{n+1/2}$  for horizontal interfaces from Godunov state
    • Update uold(:,j,:) (see equation (1))
end for

// Update on the second direction (idim=2), row by row
for i=1,nx do
    • u(i,:)=uold(i, :, :) // copy of rows of uold in the working buffer u
    • Compute primitive variables  $q(i,:) = (\rho, u, v, p)$ 
    • Solve Riemann problem at current cell interfaces, i.e compute Godunov state
    • Compute incoming fluxes  $G_{i,j+1/2}^{n+1/2}$  for vertical interfaces from Godunov state
    • Update uold(i, :, :) (see equation (1))
end for

```

Table 2. Godunov routine, computation of updated values of uold(:, :, :) at t+dt

3

Figure 1. Update of $uold(:,:,i)$ along the second dimension (row by row) computation of fluxes at all horizontal interfaces.

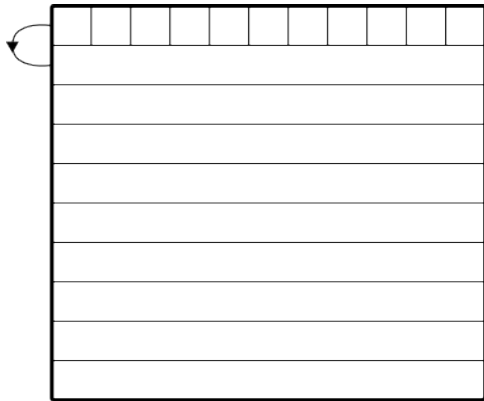
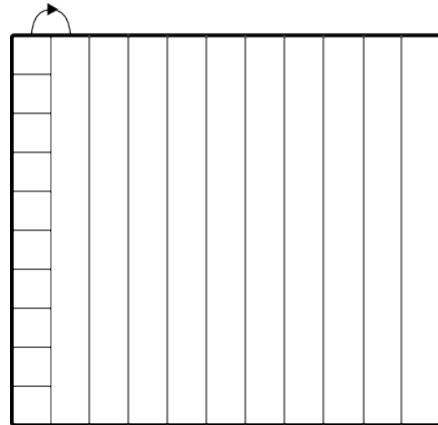


Figure 2. Update of $uold(:,i,:)$ along the first dimension (column by column), computation of fluxes at all vertical interfaces.



The Fortran Branch

3.1 Sequential version

The Fortran sequential version of the code is the reference version used to validate numerical results for any new version of the code. All results presented in this paper are validated this way.

3.1.1 Structure

The code is written in Fortran90 and includes 5 files:

- main.f90: implementation of algorithm in Table 1
- module_hydro_principal.f90 : implementation of algorithm of Table 2 and of the computation of the new time step
- module_hydro_utils.f90: implementation of Riemann solver
- module_hydro_IO.f90: implementation of read and write routines
- module_hydro_commun.f90 : parameters and constants

```

20      ! Read run parameters
21      call read_params
22
23      ! Initialize hydro grid
24      call init_hydro
25
26      print *
27      print *, ' Starting time integration, nx = ',nx,' ny = ',ny
28      print *
29
30      ! Main time loop
31      do while (t < tend .and. nstep < nstepmax)
32
33          ! Output results
34          if ( on_output .and. MOD(nstep,noutput)==0) then
35              call output
36          end if
37
38          ! Compute new time-step
39          if(MOD(nstep,2)==0) then
40              call cmpdt(dt)
41              if(nstep==0) dt=dt/2.
42          endif
43
44          ! Directional splitting
45          if(MOD(nstep,2)==0) then
46              call godunov(1,dt)
47              call godunov(2,dt)
48          else
49              call godunov(2,dt)
50              call godunov(1,dt)
51          end if
52
53          nstep=nstep+1
54          t=t+dt
55          write(*,('step=",I6," t=",1pe10.3," dt=",1pe10.3)')nstep,t,dt
56
57      end do

```

Figure 3. main program in main.f90

```

133      ! Update boundary conditions
134      call make_boundary(idim)
135
136      if (idim==1) then
137          ! Allocate work space for 1D sweeps
138          call allocate_work_space(imin,imax,nx+1)
139
140      do j=jmin+2,jmax-2
141          ! Gather conservative variables
142          do i=imin,imax
143              u(i,ID)=uold(i,j,ID)
144              u(i,IU)=uold(i,j,IU)
145              u(i,IV)=uold(i,j,IV)
146              u(i,IP)=uold(i,j,IP)
147          end do
148          if(nvar>4) then
149
150              ! Convert to primitive variables
151              call constoprims(u,q,c)
152
153              ! Characteristic tracing
154              call trace(q,dq,c,qxm,qxp,dt dx)
155
156              do in = 1,nvar
157                  do i=1,nx+1
158                      qleft (i,in)=qxm(i+1,in)
159                      qright(i,in)=qxp(i+2,in)
160                  end do
161              end do
162
163              ! Solve Riemann problem at interfaces
164              call riemann(qleft,qright,qgdnv, &
165                          rl,ul,pl,cl,wl,rr,ur,pr,cr,wr,ro,uo,po,co,wo, &
166                          rstar,ustar,pstar,cstar,sgnm,spin,spout, &
167                          ushock,frac,scr,delp,pold,ind,ind2)
168
169              ! Compute fluxes
170              call cmpflx(qgdnv,flux)
171
172              ! Update conservative variables
173              do i=imin+2,imax-2
174                  uold(i,j,ID)=u(i,ID)+(flux(i-2,ID)-flux(i-1,ID))*dt dx
175                  uold(i,j,IU)=u(i,IU)+(flux(i-2,IU)-flux(i-1,IU))*dt dx
176                  uold(i,j,IV)=u(i,IV)+(flux(i-2,IV)-flux(i-1,IV))*dt dx
177                  uold(i,j,IP)=u(i,IP)+(flux(i-2,IP)-flux(i-1,IP))*dt dx
178              end do
179              if(nvar>4) then
180
181              end do
182
183              ! Deallocate work space
184              call deallocate_work_space
185

```

Figure 4. Godunov subroutine in module_hydro_principal.f90.

3.1.2 Performance

The Figure 5 shows a profiling for the sequential version on an *IBM SP-6* architecture with the tool *gprof*. The data set used for this profiling is a square domain of size $n_x=n_y=10000$.

It appears clearly that the most intensive computing parts of the code corresponds to the two subroutines *GODUNOV* (see algorithm in Table 2) and *RIEMANN* which computes Riemann solver. They represent approximately 80% of the total consumed CPU time. This distribution with two hot spots allows anticipating that it will greatly benefit from parallelization or porting to accelerated hardware, leading to improved performance and good efficiency.

nggranularity: Each sample hit covers 4 bytes. Time: 473.43 seconds

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
42.0	198.96	198.96	80000	2.49	2.49	._hydro_utils_NMOD_riemann [4]
37.7	377.21	178.25	8	22281.25	57330.75	._hydro_principal_NMOD_godunov [3]
5.8	404.45	27.24	80000	0.34	0.47	._hydro_utils_NMOD_trace [5]
4.9	427.46	23.01	80000	0.29	0.45	._hydro_utils_NMOD_constoprism [6]
3.5	444.13	16.67	100000	0.17	0.17	._hydro_utils_NMOD_eos [7]
2.3	454.87	10.74	80000	0.13	0.13	._hydro_utils_NMOD_slope [8]
1.5	461.92	7.05	80000	0.09	0.09	._hydro_utils_NMOD_cmpflx [10]
1.5	468.85	6.93	2	3465.00	5132.00	._hydro_principal_NMOD_cmpdt [9]
0.9	473.23	4.38	1	4380.00	4380.00	._hydro_principal_NMOD_init_hydro [11]

Figure 5. Profiling of the sequential version of HYDRO on a square domain of size $n_x=n_y=10000$

We analyzed performance of the sequential version of HYDRO with the tool *hpccount*. Results are presented below. We obtained 5.8 % of peak performance with 1.08 Gflop/s sustained and more than 57% of FMA operations.

hpccount v3.2.1 (IHPCT v2.2.0) summary

Resource Usage Statistics

Total amount of time in user mode : 482.755073 seconds

Total amount of time in system mode : 0.007216 seconds

Maximum resident set size : 3133640 Kbytes

...

End of Resource Statistics

Execution time (wall clock time) : 485.332292710897 seconds

PM_FPU_1FLOP (FPU executed one flop instruction) : 212234778018

PM_FPU_FMA (FPU executed multiply-add instruction) : 151820496037

PM_FPU_FSQRT_FDIV (FPU executed FSQRT or FDIV instruction) : 10402420664

PM_FPU_FLOP (FPU executed 1FLOP, FMA, FSQRT or FDIV instruction) : 374457694719

PM_RUN_INST_CMPL (Run instructions completed) : 999174066793

PM_RUN_CYC (Run cycles) : 2272903363948

Utilization rate : 99.558 %

Instructions per run cycle : 0.440

Table 3. IBM hpccount tool

3.2 *OpenMP Fine-Grain version*

3.2.1 *Implementation*

For the OpenMP Fine-Grain approach we use OpenMP directives to share the work between the threads, especially on the level of parallel loops with the `DO` directives (e.g. looplevel parallelism with a unique parallel region). It is the classical way to parallelize an application with OpenMP.

From the profiling of the sequential version of HYDRO, we identified that all the potential acceleration is related to the parallelization of the two routines `GODUNOV` and `RIEMANN`. OpenMP philosophy is to share work at the highest level of the code to minimize overhead and optimize performance. As `RIEMANN` is called from `GODUNOV`, we created one parallel region at the main level containing the temporal loop and we shared work among the threads at `GODUNOV` level. We can see in Figure 6 an extract of `GODUNOV` subroutine and how we parallelize the algorithm contained in Table 2. The computation of the time step, routine `CMPDT` (Table 1) has also been parallelized taking care to handle properly the reduction with the OpenMP clause `!$OMP DO REDUCTION` in Figure 7.

Finally, the full parallelization of the code required only 6 OpenMP constructs for a total of 35 additional lines. We dealt with the privatization of variables declared in modules with the appropriate OpenMP `THREADPRIVATE` directive. No further restructuration of the code was needed.

3.2.2 *Advantages*

- The implementation is simple; the parallelization does not alter the code; one single version of the code is to be managed for the sequential and parallel versions.
- An incremental approach of parallelization of the code is possible.
- If we use the OpenMP directives of work-sharing (`WORKSHARE`, `DO`, `SECTION`), then implicit synchronizations managed by OpenMP greatly simplify the ease of programming (e.g. parallel loop with reduction).

3.2.3 *Disadvantages*

- The additional costs due to work-sharing and the creation/management of threads can turn out to be important, particularly when the granularity is small.
- The scalability of the code is limited and inferior to the one of the MPI version.
- The execution environment must be carefully tuned (e.g. memory affinity, bindings, ...) under penalty of seeing a dramatic decrease of performance.

```

127  ▾ subroutine godunov(idim,dt)
128      use hydro_commons
129      use hydro_const
130      use hydro_parameters
131      use hydro_utils
132      use hydro_work_space
133      implicit none
134
135      ! Dummy arguments
136      integer(kind=prec_int), intent(in) :: idim
137      real(kind=prec_real), intent(in) :: dt
138      ! Local variables
139      integer(kind=prec_int) :: i,j,in
140      real(kind=prec_real) :: dtdx
141
142      ! constant
143      dtdx=dt/dx
144
145      ! Update boundary conditions
146      !$OMP SINGLE
147      call make_boundary(idim)
148      !$OMP END SINGLE
149
150  ▾ if (idim==1) then
151
152      ! Allocate work space for 1D sweeps
153      call allocate_work_space(imin,imax,nx+1)
154
155      !$OMP DO SCHEDULE(RUNTIME)
156  ▸ do j=jmin+2,jmax-2
211      !$OMP END DO
212
213      ! Deallocate work space
214      call deallocate_work_space
215
216  ▸ else
283

```

Figure 6. Subroutine GODUNOV - OpenMP Fine-Grain version

```

77  subroutine cmpdt(dt)
78      use hydro_commons
79      use hydro_const
80      use hydro_parameters
81      use hydro_utils
82      implicit none
83
84      ! Dummy arguments
85      real(kind=prec_real), intent(out) :: dt
86      ! Local variables
87      integer(kind=prec_int) :: i,j
88      real(kind=prec_real) :: cournox=0,cournoy=0,eken
89      real(kind=prec_real), dimension(:,,:), allocatable :: q
90      real(kind=prec_real), dimension(:) , allocatable :: e,c
91
92      ! compute time step on grid interior
93      cournox = zero
94     ournoy = zero
95      !$OMP BARRIER
96
97      allocate(q(1:nx,1:IP),e(1:nx),c(1:nx))
98
99      !$OMP DO REDUCTION(MAX:cournox,cournoy)
100     do j=jmin+2,jmax-2
101
102     do i=1,nx
103
104         call eos(q(1:nx,ID),e,q(1:nx,IP),c)
105
106         cournox=max(cournox,maxval(c(1:nx)+abs(q(1:nx,IU))))
107        ournoy=max(cournoy,maxval(c(1:nx)+abs(q(1:nx,IV))))
108
109     end do
110     !$OMP END DO
111
112     deallocate(q,e,c)
113
114     !$OMP SINGLE
115     dt = courant_factor*dx/max(cournox,cournoy,smallc)
116     !$OMP END SINGLE
117 end subroutine cmpdt

```

Figure 7. Subroutine CMPDT - OpenMP Fine-Grain version

3.3 OpenMP Coarse-Grain version

3.3.1 Implementation

The philosophy of this approach is similar to the one used for classical MPI domain decomposition. Each thread works on its own sub-domain (SPMD) and the synchronizations is managed carefully so that the semantic of the code is preserved. Given the initial grid ($n_x \times n_y$ points), we divide it into as many sub-domains as number of threads and map one sub-domain to one thread. There is no overlapping on sub-domains and each sub-domain is defined by both its lower boundary (`iminloc`) and upper boundary (`imaxloc`) on the first direction and by its lower boundary (`jminloc`) and upper boundary (`jmaxloc`) on the second direction (see Figure 8 for an example on a 10x10 grid with 6 threads).

This has been implemented in a sixth Fortran file `omp_domain_decomposition.f90` and added to the Fortran

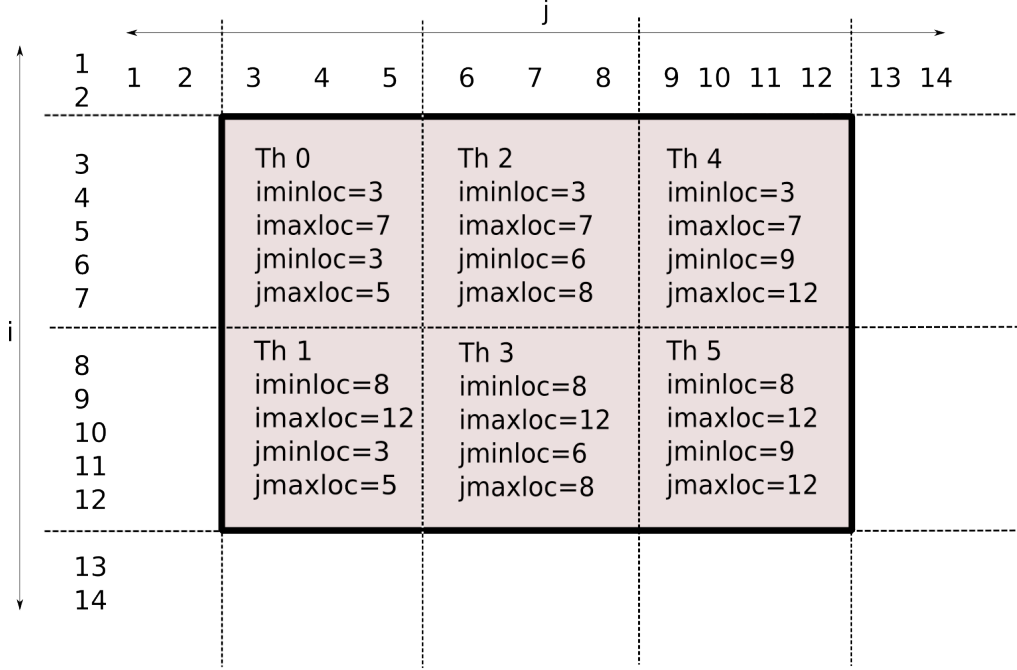


Figure 8 – OpenMP Coarse-Grain domain decomposition of a domain of size $n_x=n_y=10$ on 6 threads

package to compute the work distribution.

For the computation of the time step, we have to deal by hand with two reductions on shared variables `cournox` and `cournoy` with the operator `MAX`. For this, we introduce two new private variables `cournox_loc` and `cournoy_loc` to compute local maximum values in parallel and then use an `ATOMIC` OpenMP construct to update atomically the final results for the shared variables `cournox` and `cournoy`.

The most complex part of the parallelization is the update of `uold` from `t` to `t+dt`. It has to be managed very carefully. In fact, if we want to preserve the semantic of the code (e.g. correctness of the parallel computed results), then synchronizations have to be added so that no dependencies are broken. To cope with this problem, we implement a software pipelining algorithm with low level synchronizations using the `FLUSH` OpenMP directive (routine `syn_x` (resp. `syn_y`) to manage synchronization when updating the domain along the first (resp. second) dimension). A complete description of this algorithm can be found in [3].

As for the Fine-Grain version, we define a unique parallel region at main level containing the temporal loop.

3.3.2 *Advantages*

- This approach minimizes work-sharing overhead, leading to improved performance and better cache use.
- On a shared memory node, we obtain a very good scalability comparable and even so often better than the one obtained by parallelizing the code with MPI.
- This is the best parallel version in terms of performance on an SMP node.

3.3.3 *Disadvantages*

- We lose the advantages of OpenMP: it is a very intrusive approach. It is no longer possible to have one and unique version of the code to manage.
- The incremental approach of code parallelization is no longer possible.
- The synchronizations (global or thread-level) are completely the programmer's responsibility.
- The work-sharing and the load balancing is also the programmer's responsibility.
- Finally, the implementation turns out to be at least as complex as a parallelization with MPI.
- The execution environment must be carefully tuned (e.g. memory affinity, bindings, ...) under penalty of seeing a dramatic decrease of performance.

```

22  ! Read run parameters
23  call read_params
24
25  !$OMP PARALLEL
26  ! Initialize hydro grid
27  call init_hydro
28
29  !$OMP SINGLE
30  print *
31  print *, 'HYDRO_OMP2D_Sync - Execution with ', omp_get_num_threads(), ' threads'
32  print *, 'On first dimension : ', dims(1), ' thread(s)'
33  print *, 'On second dimension : ', dims(2), ' thread(s)'
34  print *
35  print *, 'Starting time integration with nx = ', nx, ' ny = ', ny
36  print *
37  deallocate(dims)
38  !$OMP END SINGLE
39
40  ! Main time loop
41  do while (t < tend .and. nstep < nstepmax)
42
43      ! Output results
44      if (on_output .and. MOD(nstep, noutput) == 0) then
45          !$OMP SINGLE
46          call output
47          !$OMP END SINGLE
48      end if
49
50      ! Compute new time-step
51      if (MOD(nstep, 2) == 0) then
52          call cmpdt(dt)
53          !$OMP SINGLE
54          if (nstep == 0) dt = dt/2.
55          !$OMP END SINGLE
56      endif
57
58      ! Directional splitting
59      if (MOD(nstep, 2) == 0) then
60          call godunov(1, dt)
61          call godunov(2, dt)
62      else
63          call godunov(2, dt)
64          call godunov(1, dt)
65      end if
66
67      !$OMP SINGLE
68      nstep = nstep + 1
69      t = t + dt
70      write(*, '( "step=", I6, " t=", 1pe10.3, " dt=", 1pe10.3 )' ) nstep, t, dt
71      !$OMP END SINGLE
72

```

One thread works and the others wait

Figure 9. main program in OpenMP Coarse-Grain version

```

180  call make_boundary(idim)
181
182  if (idim==1)then
183      ! Allocate work space for 1D sweeps
184      call allocate_work_space(iminloc-2,imaxloc+2)
185
186  do j=iminloc,jmaxloc
187      ! Gather conservative variables
188      do i=iminloc-2,imaxloc+2
189          if(nvar>4)then
190
191              ! Le thread num_th a lu la jieme ligne de son sous-domaine
192              ! Ce flush est necessaire pour assurer que la lecture de uold
193              ! est terminee avant de mettre a jour tab_sync, sinon le compilateur
194              ! pourrait intervertir les deux traitements !
195              !$OMP FLUSH(tab_sync,u)
196              tab_sync(num_th)=j
197              !$OMP FLUSH(tab_sync)
198
199              ! Convert to primitive variables
200              call constoprime(u,q,c)
201
202              ! Characteristic tracing
203              call trace(q,dq,c,qxm,qxp,dt dx)
204
205              do in = 1,nvar
206
207                  ! Solve Riemann problem at interfaces
208                  call riemann(qleft,qright,qgdv, &
209                      rl,ul,pl,cl,wl,rr,ur,pr,cr,wr,ro,uo,po,co,wo, &
210                      rstar,ustar,pstar,cstar,sgnm,spin,spout, &
211                      ushock,frac,scr,delp,pold,ind,ind2)
212
213                  ! Compute fluxes
214                  call cmpflx(qgdv,flux)
215
216                  ! MAJ ssi voisin lecture terminee...
217                  call sync_x
218
219                  do i=iminloc,imaxloc
220                      if(nvar>4)then
221
222                          end do
223                          !$OMP BARRIER
224
225                  ! Deallocate work space

```

Private subdomain

Manual flushes and synchronisations

Figure 10. Godunov subroutine in OpenMP Coarse-Grain version

3.4 *MPI version*

3.4.1 *Implementation*

The MPI version is based on 2D domain decomposition with ghost (or halo) cells. It uses derived MPI datatypes and there is no computation/communication overlap. We added to the code a sixth file `mpi_module.f90` which contains MPI parameters and MPI specific routines. It acts like the additional file `omp_domain_decomposition.f90` in the OpenMP Coarse-Grain version.

The routine `Godunov` is very similar to the one of the OpenMP Coarse-Grain version, but with a local numbering.

For sake of simplicity, we decided to introduce a new type of boundary condition to deal with the update of halo cells of each subdomain at each time step. This new MPI boundary condition is implemented in the routine `make_boundaries` (see Table 2) using `MPI_SENDRECV` for point-to-point neighbourhood communication scheme (see Figure 11).

For the computation of the time step in routine `CMPDT` (see Table 1), we use the collective function `MPI_ALLREDUCE` to deal with the global reduction across MPI processes.

3.4.2 *Advantages*

- The MPI version is perfectly well-balanced and the communication scheme is nearly optimal as it involves only neighborhood point-to-point communications (`MPI_SENDRECV`) except for the time step computation which involves a global reduction (`MPI_ALLREDUCE`).
- As most of the work of the MPI implementation was to carefully define the topology, the domain decomposition and the MPI derived type in module `mpi_module.f90`, the core of the code remains unchanged.

3.4.3 *Disadvantages*

- It is an intrusive approach, but changes are limited to a few well-defined parts of the code.
- We find the same disadvantages as those described in section 3.3.3.

```

19
20 ▾ subroutine make_boundary(idim)
21     use hydro_mpi
22     use hydro_commons
23     use hydro_const
24     use hydro_parameters
25     implicit none
26
27     ! Dummy arguments
28     integer(kind=prec_int), intent(in) :: idim
29     ! Local variables
30     integer(kind=prec_int) :: ivar,i,i0,j,j0
31     real(kind=prec_real) :: sign
32     !!$ integer(kind=prec_int) :: ijet
33     !!$ real(kind=prec_real) :: djet,ujet,pjet
34
35 ▾ if(idim==1) then
36
37     !--MPI--!
38     ! Send to east and receive from west
39     call MPI_SENDRECV(uold(nx+1,3,1),1,bloc_dim1,voisins(EAST),etiquette, &
40                      uold(1,3,1),1,bloc_dim1,voisins(WEST),etiquette, &
41                      comm2d,MPI_STATUS_IGNORE,code)
42
43     ! Send to west and receive from east
44     call MPI_SENDRECV(uold(3,3,1),1,bloc_dim1,voisins(WEST),etiquette, &
45                      uold(nx+3,3,1),1,bloc_dim1,voisins(EAST),etiquette, &
46                      comm2d,MPI_STATUS_IGNORE,code)
47
48     !--MPI--!
49
50     ! Left boundary
51     if (boundary_left>0) then
52
53     ! Right boundary
54     if (boundary_right>0) then
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70 ▾ else
71
72     !--MPI--!
73     ! Send to south and receive from north
74     call MPI_SENDRECV(uold(3,3,1),1,bloc_dim2,voisins(SOUTH),etiquette, &
75                      uold(3,ny+3,1),1,bloc_dim2,voisins(NORTH),etiquette, &
76                      comm2d,MPI_STATUS_IGNORE,code)
77
78     ! Send to north and receive from south
79     call MPI_SENDRECV(uold(3,ny+1,1),1,bloc_dim2,voisins(NORTH),etiquette, &
80                      uold(3,1,1),1,bloc_dim2,voisins(SOUTH),etiquette, &
81                      comm2d,MPI_STATUS_IGNORE,code)
82
83     !--MPI--!
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101

```

Figure 11. Subroutine make_boundary - MPI version

3.5 Performance of OpenMP Fine-Grain, OpenMP Coarse-Grain and MPI version

We compare the performance and the parallel strong scaling scalability of the three versions of HYDRO: OpenMP Fine-Grain version, OpenMP Coarse-Grain version and MPI version.

3.5.1 Target machine characteristics

The target machine is an IBM SP6 shared memory node with 32 cores per node.

3.5.2 Characteristics of used data sets

We use three data sets, of the same total size (i.e the same total number of points), but with different distributions in the two directions x and y :

- elongated domain in the y direction: $n_x=1000$, $n_y=100000$ (see figure 12),
- square domain in the y direction: $n_x=n_y=10000$ (see figure 13),
- elongated domain in the x direction: $n_x=100000$, $n_y=1000$ (see figure 14),

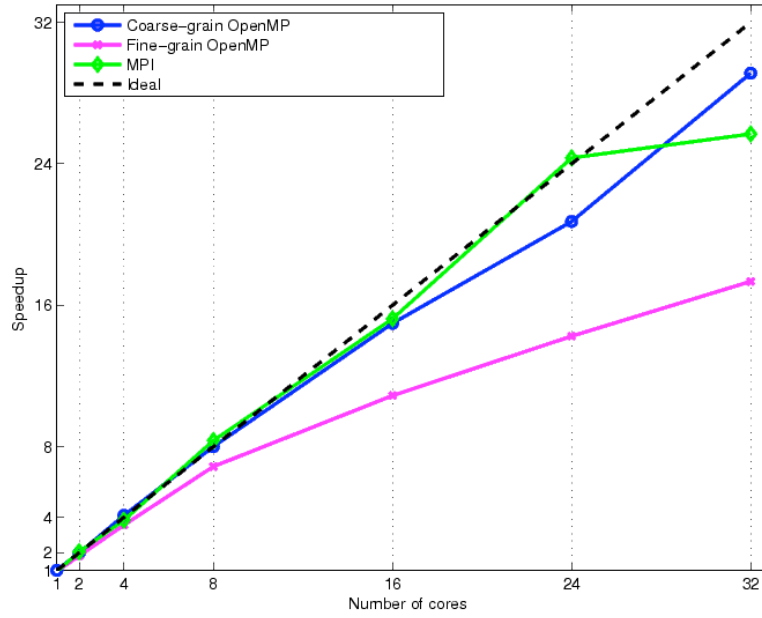


Figure 12. Strong scaling test on an elongated domain in y direction: $n_x=1000$ and $n_y=100000$

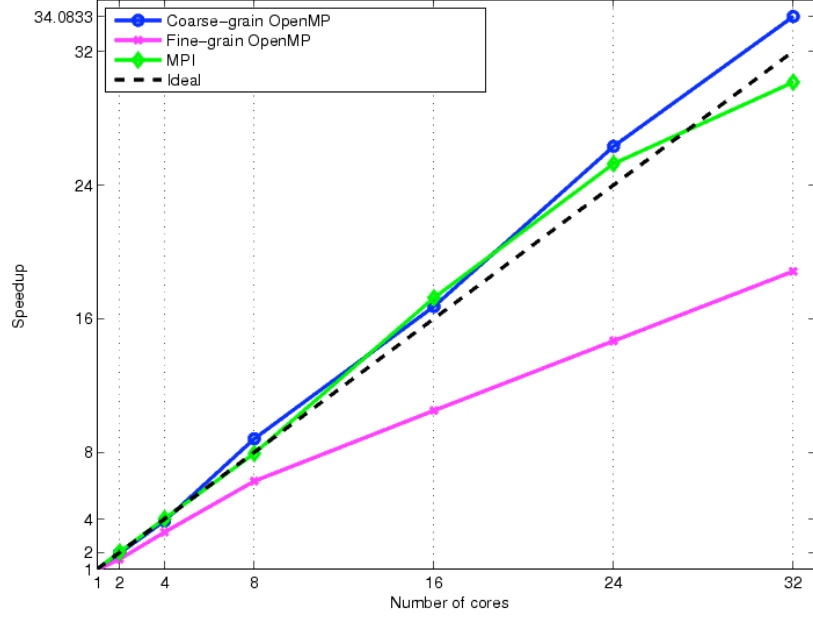


Figure 13. Strong scaling test on a square domain: $n_x=n_y=10000$

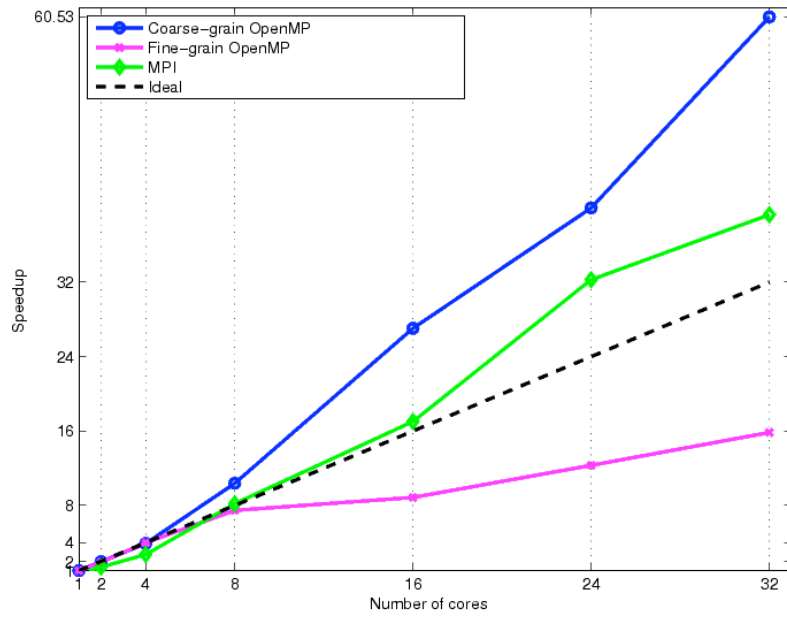


Figure 14. Strong scaling test on an elongated domain in x direction: $n_x=100000$ and $n_y=1000$

3.5.3 *Analysis of results*

- Although we work on domains of identical size, the obtained performance on one core vary from simple to the triple. It comes from the good use or not of caches.
- Whatever the data set and up to 4 cores, the three versions give more or less similar performance. Beyond 4 cores, the OpenMP Fine-Grain version suffers from a problem of degraded scalability compared to the MPI and OpenMP Coarse-Grain versions.
- Accordingly, except for a limited number of cores, the OpenMP Fine-Grain version is always largely dominated by the MPI or OpenMP Coarse-Grain versions, even if it is linearly scalable on a limited way, up to 32 cores.
- Up to 24 cores, the MPI and OpenMP Coarse-Grain versions have similar and perfect scalability, sometimes even super-linear (i.e. better re-use of caches as the size of the local sub-domains decreases).
- Beyond 24 cores, the MPI version seems to slow down, while the OpenMP Coarse-Grain version continues to be perfectly scalable.
- On 32 cores, it is always the OpenMP Coarse-Grain version that gives the best results.
- It is important to say that the MPI version can still be optimized by implementing computation/communication overlap, which could enable it to be scalable beyond 24 core.

3.5.4 *Conclusions*

- The generalization of shared-memory machines as well as the increase of number of cores available inside a node requires that we reconsider the way we parallelize applications.
- Inside a node, if we look for the maximum performance, it is the OpenMP Coarse-Grain approach that must be used. This requires a great investment and it is at least as much complicated to implement as an MPI version. The debugging is particularly complex. This approach is reserved to specialists who master skillfully the parallelism and its traps.
- The simplicity of use and the rapidity of implementation of an OpenMP Fine-Grain version are its main advantages. On the condition of well coding, performance according to the type of algorithm (especially according to the granularity) can range from medium to relatively good. The debugging still remains complex. This approach is destined to everyone.
- MPI obtains good performance on a shared-memory node, but the OpenMP Coarse-Grain version outclasses it in terms of scalability and performance. It can still however be optimized, for example by implementing the overlapping of computation by communications. It stays anyway indispensable when it is necessary to go beyond the use of a single node.

3.6 Hybrid MPI/OpenMP version

3.6.1 Implementation

Starting from both a pure flat MPI and an OpenMP parallel version that are already available, we combine the two approaches to create a hybrid MPI+OpenMP version of HYDRO. For reasons of portability and simplicity, the `MPI_THREAD_FUNNELED` level of thread support is used. Porting the code to the MPI+OpenMP paradigm turns out to be straightforward; the additional porting/coding effort being very limited compared to the time needed to develop the pure MPI or the OpenMP versions. We used here a 2D domain decomposition for the MPI level and a Coarse-Grain OpenMP parallelization. From our experience and as the OpenMP Coarse-Grain version has been very intrusive and complex to implement, we decided to fuse the two versions by incorporating MPI calls to the OpenMP version.

3.6.2 Characteristics of used data sets

We built two data sets, much bigger than those used so far in order to run the code on the PRACE Research Infrastructure Tier-0 systems, in particular CURIE (TGCC, France) and JUGENE (JSC, Germany).

- A data set for strong scaling runs:
 - Number of points of the domain: $N_x * N_y = 40000 * 40000 = 1.6 * 10^{+09}$,
 - Number of iterations: 4
 - Memory used: 8Gb (90Mb/core on 4096 cores),
 - Elapsed time on 4096 cores BG/P: about 8 seconds,
 - Number of targeted cores: 4K, 8K, 16K, 32K
- A data set for weak scaling runs as provided in Table 4.

Cores	$N_x * N_y$	Iterations	Memory/core (Mb)
4096	40000^2	4	90
8192	56568^2	4	90
16384	80000^2	4	90
32768	131137^2	4	90

Table 4. Weak scaling data set

3.6.3 Results of strong scaling runs

The gain in terms of performance or scalability by using a hybrid version is not obvious. In fact, as already mentioned, the MPI version is perfectly well-balanced and the communication scheme is nearly optimal as it involves only neighbourhood point-to-point communication except for the time step computation which involves a global reduction.

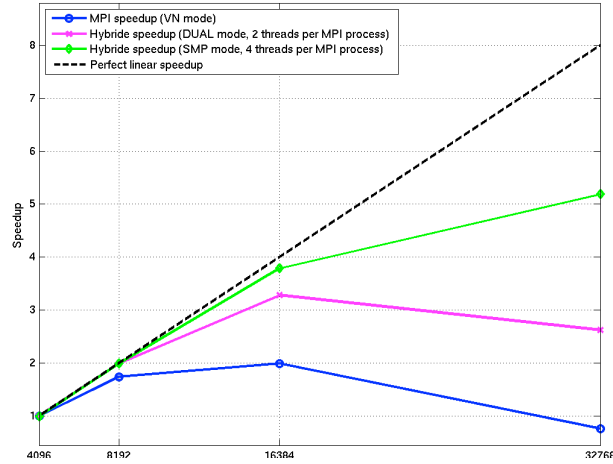


Figure 15. Strong scaling tests on Blue Gene/P

- The strong scaling results presented here were run on the Blue Gene/P at Jülich (Germany).
- From small to moderate number of cores, the performance of pure MPI or MPI+OpenMP approaches are very similar.
- Beyond 4096 cores, the pure MPI implementation begins to lose scalability, whereas the hybrid approach keeps a near perfect scalability up to 16384 cores and even continues to scale up (non-linearly) to 32768 cores.
- On this strong scaling test, the scalability limit of the flat MPI version is 8192 cores, whereas the scaling limit of the SMP hybrid version is 32768 cores. We find here the factor of four which correspond to the number of cores of a BG/P node.
- The best hybrid version (on 32768 cores) is 3.5 times faster than the best MPI version (on 8192 cores).

3.6.4 Results of weak scaling runs

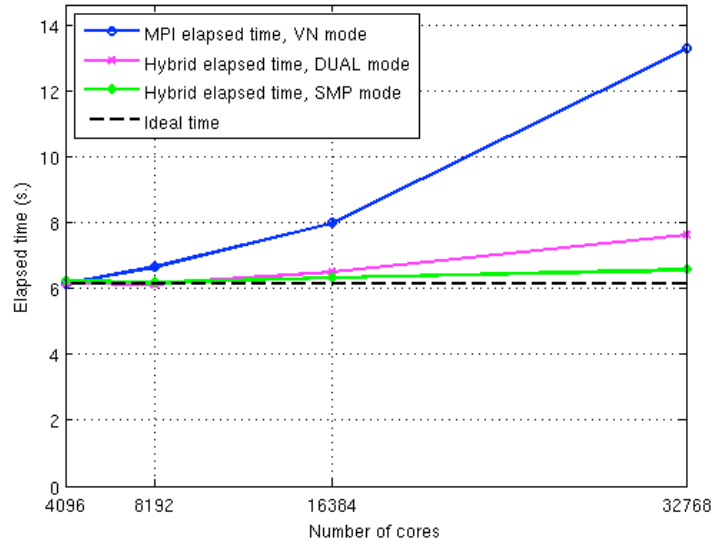


Figure 16. Weak scaling tests on Blue Gene/P

- The scalability of the MPI flat version shows its limits. It hardly scales up to 16384 cores, then the elapsed time explodes beyond that.
- The DUAL hybrid version (2 threads per MPI process), but, even more, the SMP version (4 threads per MPI process) behaves well up to 32768 cores, with nearly constant elapsed time.
- In this weak scaling test, the scalability limit of the MPI flat version is 16384 cores. The limit of the SMP version is not yet reached on 32768 cores.
- It is clear that with this type of parallelization method (i.e. domain decomposition), the scaling (here over 16K cores) clearly requires the use of hybrid parallelization.

3.6.5 *Learning from experiences - Advantages*

- Hybrid parallel programming consists of mixing the MPI parallel paradigm with a threaded parallel model in order to benefit from the advantages of both approaches.
- Compared to a flat MPI approach, the hybrid parallel approach decreases the number of MPI processes leading to the following consequences:
 - Better scalability due to a reduction of both the number of MPI messages, and the number of processes involved in collective communication (MPI_ALLTOALL is not very scalable)
 - Better granularity at MPI level and better load balancing at the node level managed implicitly by OpenMP using the shared memory
 - Better adequacy to the architecture of modern supercomputers (interconnected shared-memory nodes, NUMA machines...), where the use of HyperThreading (HT) or Simultaneous Multi Threading (SMT) will play an even more important key role in the future in order to achieve efficient performance
 - Memory savings
 - The ghost or halo cells, introduced in order to simplify the programming of MPI codes using domain decomposition, are no longer mandatory within the SMP node. Only the ghost cells associated with the inter-node communications are mandatory.
 - The memory footprint of system buffers associated with MPI is not negligible and increases with the number of processes. For example, for an Infiniband network with 65000 MPI processes, the memory footprint of system buffers reaches 300MB per process, almost 20TB in total!
- Removal of some algorithmic limitations (maximum decomposition in one direction for example) which concretely limit the total number of MPI processes.
- Enhancement of the efficiency of some algorithms: fewer bigger domains may mean a better preconditioner if we drop the contributions of other domains.
- Optimization of I/O.
 - Fewer simultaneous accesses in I/O and larger average record size. This causes less load on the meta-data servers with requests of more suitable size. The potential saving for a massively parallel application can be significant.
 - Fewer files to manage if we use an approach where the number of files is proportional to the number of MPI processes (an approach not at all recommended in a context of massive parallelism).

3.6.6 *Learning from experiences – Disadvantages*

Managing parallelism at two different levels is hard to implement and error prone. Thread synchronisation, definition of variable status (SHARED or PRIVATE), race condition are common pitfalls to avoid.

Debugging is especially difficult as tools are not mature.

All the disadvantages listed previously for MPI and OpenMP are still valid.

Most advanced implementations of hybrid parallelism using the MPI_THREAD_MULTIPLE level of threads support are totally inefficient on all tested architectures.

```

5 program hydro_main
6   use hydro_commons
7   use hydro_parameters
8   use hydro_io
9   use hydro_mpi
10  use hydro_principal
11  use Ptim
12  use omp_parameters
13  use omp_domain_decomposition
14  implicit none
15
16  real(kind=prec_real) :: dt
17
18  !--MPI--!
19  ! Initialization of MPI environment
20  call MPI_INIT_THREAD(MPI_THREAD_FUNNELED,level_mpi_provided,code)
21  ! MPI timing
22  call PTIM_start(LABEL="Hydro MPI-OpenMP")
23  !--MPI--!
24
25  ! Read run parameters
26  call read_params
27
28  ! Initialize MPI domains and communicator
29  call init_mpi
30
31  !$OMP PARALLEL
32  ! Initialize OpenMP domains
33  call init_omp
34
35  if (rang==0) then
36    !$OMP SINGLE
37    deallocate(dims)
38    !$OMP END SINGLE
39
40    ! Initialize hydro grid
41    call init_hydro
42    !print *, "Fin init_hydro"
43
44    ! Main time loop
45    do while (t < tend .and. nstep < nstepmax)
46
47      ! Output results
48      if ( on_output .and. MOD(nstep,noutput)==0) then
49        !$OMP SINGLE
50        call output
51        !$OMP END SINGLE
52      end if
53
54      ! Compute new time-step
55      if (MOD(nstep,2)==0) then
56        call cmpdt(dt)
57        !$OMP SINGLE
58        if (nstep==0) dt=dt/2.
59        !$OMP END SINGLE
60      endif
61
62      ! Directional splitting
63      if (MOD(nstep,2)==0) then
64        call godunov(1,dt)
65        call godunov(2,dt)
66      else
67        call godunov(2,dt)
68        call godunov(1,dt)
69      end if
70
71      !$OMP SINGLE
72      nstep=nstep+1
73      t=t+dt
74      !$OMP END SINGLE
75
76      !--MPI--!
77      if (rang==0 .and. num_th==0) write(*,('step=",I6," t=",1pe10.3," dt=",1pe10.3')nstep,t,dt)
78      !--MPI--!
79
80    end do
81  end if
82
83  end program hydro_main

```

Figure 17. Main program of the hybrid MPI/OpenMP version

4 The C branch

When this work was started back in 2009, the only operational way to have a code that could be written in CUDA, OpenCL or HMPP was using the C89 language. At that time CUDA couldn't handle C++, OpenCL relied (and relies) on C99 and the maturity of FORTRAN HMPP wasn't good enough for a decent comparison.

Since those days, the landscape has evolved quite a bit for both CUDA and HMPP. Cuda is getting closer to a full support of C++ (since the Fermi hardware is helping a lot), HMPP is now mature enough to cope with most of the FORTRAN constructs alongside with C99.

For the aforementioned reasons, we did the port in C89 to start with. Then, from this base, we implemented a CUDA version. It was then easy to do the OpenCL and HMPP ports. The move to C99 was the opportunity to implement a variant of the initial algorithm and jump to the MPI world. Once the MPI version was implemented, it was easy to back port both the new algorithm and the MPI communications in the CUDA version. The following sections will follow this time track, trying to capture what has been learned at each step.

4.1 C89

This first port was done with the constraint of keeping as much as possible the structure and naming conventions of the initial sequential FORTRAN code. The goal wasn't to improve the numerical method but rather to compare languages in as close forms as possible. The final result reflects quite faithfully the original. Yet some important changes were required to prepare the next phase: the CUDA port.

4.1.1 2D Arrays

C89 is a very reliable but crude language. nD arrays are not available with respect to the FORTRAN habits. We had to revert to using macros to implement 2D indexation. This leads to a rather cumbersome way of coding which has the drawback of being difficult to debug.

```
#define IHVW(i, v) ((i) + (v) * Hnxyt)
r = q[IHVW(i, ID)];
```

Table 5. Array indexing

The advantage of such a coding is that the FORTRAN indexing scheme can be kept, leading to a code looking pretty much like the original. The drawback is to use a different macro for each type of array if we don't want to add extra parameters (the leading dimension of the array in bold in the snippet above). This introduces also the constraint of a consistent naming of the variables holding dimensions. While highly desirable, former experiments show that it is not always the case.

The FORTRAN 90 array syntax hides a lot of loops and even auxiliary memory usage. It is therefore required to avoid this feature in favor of actual loops to ease the port to CUDA.

4.1.2 Arguments to SUBROUTINES

One of the drawbacks of C89 is that we can't specify the array dimensions as easily as in FORTRAN. This feature called Variable Length Array is available only in C99 (and not in C++ even). Therefore one can't write such a declaration in C89:

```
void foo(int n, double arr[n]) { /* my code */ };
```

Table 6. A VLA declaration in C99

This syntax is (unfortunately) invalid in C89.

To circumvent such a limitation we rely on both explicit dimensions in the argument list and on the `#define` shown previously for the array indexing. One should note that checking of overflow is not available through this method (a FORTRAN compiler is able to do it since dimensions are used, not pointers).

```
Void
trace(double *RESTRICT q, double *RESTRICT dq, double *RESTRICT c,
       double *RESTRICT qxm, double *RESTRICT qxp,
       const double dtdx, int n,
       const int Hscheme, const int Hnvar, const int Hnxyt)
{ // local variables
  // ...
#define IHVW(i, v) ((i) + (v) * Hnxyt)
// ...
```

Table 7. 2D Arrays

This section of code illustrates how we deal with array dimensions.

Whenever possible the usage of `const` and `restrict` is enforced to help the compiler.

Another pattern we used was to forbid global variables altogether. This rule was introduced to get prepared for the CUDA port. It means that all arrays had to be passed as arguments to each function as illustrated above for the `trace()` routine. Note that a FORTRAN `USE` has the same effect as using C global variables and thus should be avoided for other usage than type declarations.

4.1.3 MODULE and USE

Since the original was clean FORTRAN 90, we had to imagine a form of reproducing the MODULE/USE construct while sticking to the C habits. We had also to get prepared to the GPU port. The solution was first to turn MODULE declarations in C structures, defined in ".h" files. Then we studied the code to isolate sections where the structures are modified. In that case, pointers to the actual structure are used. Everywhere else, the value of the structure is used to avoid side effects on pseudo global values.

Furthermore, we decided that none of the structures could be used directly in subroutines but the main one. Therefore structure members should always be passed as arguments (either by value or pointer depending on the usage). A pure computation routine will access only to members, prohibiting any change in the main structure.

```

typedef struct _hydroparam {
    int prt;
    // time control
    double t, tend;
    int nstep, nstepmax;
    int noutput;
    double dtoutput;
    // dimensions
    int imin, imax, jmin, jmax, nx, ny, nxt, nyt, nxyt;
    // physics
    int nvar;
    double dx;
    double gamma;
    double courant_factor;
    double smallc, smallr;
    // numerical scheme
    int niter_riemann;
    int iorder;
    double slope_type;
    int scheme;
    int boundary_right, boundary_left, boundary_down, boundary_up;
} hydroparam_t;
hydroparam_t H;

int main(int argc, char **argv) {
    // ...
    hydro_init(&H, &Hv);
    // ...
}

```

Table 8. Module transformation

This snippet of code illustrates how a MODULE has been transformed in a structure (hydroparam_t) and initialized (by address).

```

if ((H.nstep % 2) == 0) {
    hydro_godunov(1, dt, H, &Hv, &Hw, &Hvw);
} else {
    hydro_godunov(2, dt, H, &Hv, &Hw, &Hvw);
}

```

Table 9. Upper level usage of the structure

Here we show how the previous structure (H) is used as a read only variable (by value) preventing any further modification (upper level of call).

<pre>trace(q, dq, c, qxm, qxp, dtdx, Hdimsize, H.scheme, H.nvar, H.nxyt);</pre>
--

Table 10. Lower level usage of the structure

Deeper inside the code, we don't pass the whole structure but only the relevant fields to simplify the signature of the function and limit side effects.

4.1.4 *Lessons learned*

- The architecture of the code should not include computations. The latter must be encapsulated in lower-level functions
- In a FORTRAN code, MODULE/USE should be used only at the upper levels of the architecture. Everything else should be arguments.
- Dimensions should always be explicit (FORTRAN77 type declaration)
- FORTRAN90 array syntax hides a lot of parallelism (and pitfalls). Prefer explicit loops.
- Dynamic memory allocation should be done at the highest level possible and be forbidden in a computational routine.

With these modifications, it was pretty straightforward to move from FORTRAN to C.

4.2 CUDA 1D

Once the C89 code was running, meaning that it gave the very same results as the FORTRAN version, we started to port it to CUDA. To achieve this, we had to study the array usage, adapt some algorithms of the code and move subroutines to CUDA. We describe each step in the following subsections.

4.2.1 Array Usage

Since we had made explicit array usage through arguments, it was easy to track the real usage of every single array. We were able to sort them as temporaries needed only on the GPU or fundamental arrays which needed to be copied back to the CPU, depending on the global algorithm of the code. This process was very useful to limit the data movements from/to the GPU as well as to create all the corresponding buffers in one shot on the GPU (`cuAllocOnDevice()`).

Note that if we had been able to stick to FORTRAN, the usage of FORTRAN90's `INTENT` would have helped us significantly by clearly stating the purpose of each array, if specified as an argument. Otherwise, using only the `USE` clause couldn't give us such an important piece of information.

To make sure of which address space we are using, a coding pattern has been used of adding a `DEV` extension to variable names representing a buffer on the GPU. Doing so, we avoided a common CUDA bug where pointers get mixed up leading to crashes.

<pre>cuTrace(qDEV, dqDEV, cDEV, qxmDEV, qxpDEV, dtdx, H.nxt, H.scheme, H.nvar, H.nxyt);</pre>

Table 11. Naming convention

A call to “trace” using the CUDA implementation. Note that the function name has been changed to allow for the cohabitation of the original C89 implementation and the CUDA port.

4.2.2 Algorithm change

The main difficulty was located in the `Riemann()` routine. In this function, the original code was using a set of indexes and arrays to process only the elements which haven't converged.

```

n = nface
do i = 1,n
    ind(i)=i
end do
! Newton-Raphson iterations to find pstar at the required accuracy
do iter = 1,niter_riemann
    do i=1,n
        wwl=sqrt(cl(ind(i))*(one+gamma6*(pold(i)-pl(ind(i)))/pl(ind(i))))
! computations skipped here!
    end do
    n_new=0
    do i=1,n
        if(uo(i)>1.d-06)then
            n_new=n_new+1
            ind2(n_new)=ind (i)
            po (n_new)=pold(i)
        end if
    end do
    j=n_new
    do i=1,n
        if(uo(i)<=1.d-06)then
            n_new=n_new+1
            ind2(n_new)=ind (i)
            po (n_new)=pold(i)
        end if
    end do
    ind (1:n)=ind2(1:n)
    pold(1:n)=po (1:n)
    n=j
end do
do i=1,nface
    pstar(ind(i))=pold(i)
end do

```

Table 12.The original FORTRAN code

One can note that the usage of the array `ind` and `ind2` makes it difficult to parallelize this algorithm on a GPU in an SIMT fashion.

To get prepared for the CUDA version, we had to simplify this algorithm. The first step was to replace the indexing mechanism with a simpler flag stating whether the algorithm has converged for the given element or not.

```
// Newton-Raphson iterations to find pstar at the required accuracy
for (iter = 0; iter < Hniter_riemann; iter++) {
    double precision = 1.e-6;
    for (i = 0; i < nface; i++) {
        if (ind[i] == 1) {
// computations skipped here!
            uo[i] = DABS(delp[i] / (pold[i] + smallpp));
            if (uo[i] <= precision) {
                ind[i] = 0;          // don't consider this cell anymore
            }
        }
    }
    // iter_riemann
for (i = 0; i < nface; i++) {
    pstar[i] = pold[i];
}
```

Table 13. A simpler version of the same algorithm

This simple transformation allows us to exhibit a first version of parallelism where the

```
for (i = 0; i < nface; i++)
```

can be implemented on the GPU since all iterations are independent. A second transformation can be immediately envisioned: permute the two for loops to avoid launching the inner kernel `Hniter_riemann` times. Then, `ind` should not be an array anymore but could be a register. This transformation fits well with the loop fusion described in the next section.

4.2.3 CUDA subroutines

Whenever a code has to be implemented in CUDA, the problem of indexation arises. We chose to always use a 1D index scheme. It allows us to use helper functions to prepare the computational grid as well as to find the element on which a kernel is working. Henceforth we can have an implementation pattern which is applied to all the computational functions.

```
SetBlockDims(((ijmax - 1) - (ijmin + 1)), THREADSSZs, block, grid);
Loop1KcuTrace <<< grid, block >>> (q, dq, c, qxm, qxp, dtdx, Hnxyt, ijmin,
ijmax, zeror, zerol, project);
CheckErr("Loop1KcuTrace");
cudaThreadSynchronize();
```

Table 14. Pattern of CUDA porting

Here we illustrate the implementation pattern of a function ported to CUDA. The grid is always computed using `SetBlockDims()` then the kernel is called and the code waits for its completion.

The easiest targets for CUDA parallelism are the loops. An automated tool (such as HMPP) would transform each single loop into a kernel. It means that the overhead of kernel calls would become important for a large number of loops. Furthermore, having numerous small loops doesn't promote high compute intensity (defined as the ratio of floating point operation by memory access). Experiments show that a compute intensity of 2 is the lowest value to benefit from a GPU. To increase the compute intensity within a loop as well as reduce the number of kernels, one has to do loop fusion (or merge) manually. This transformation has been very useful in the `riemann()` function where most of the code has been put into a single loop over the faces. This was fully compatible and made possible by the transformation of the algorithm described earlier. As a result, Riemann is implemented in a single kernel as shown in the next figure.

```
__global__ void Loop1KcuRiemann() {
    long tid = idx1d1();
    long i = idx1d();
    if (i >= K.narray) return;
// ...
    for (iter = 0; iter < K.Hniter_riemann; iter++) {
        double precision = 1.e-6;
// ...
        indi = uoS > precision;
        if (!indi)
            break;
    }
// ...
    if (K.sgnm[i] == 1) {
        K.qgdnv[IHVW(i, IV)] = K.qleft[IHVW(i, IV)];
    }
}
```

Table 15. The kernel implementing Riemann

Most of the computations have been stripped out for the sake of clarity. `idx1d1` and `idx1d` are helper functions that map CUDA indexing to our problem's index.

The final difficulty, while porting the code to CUDA, was to implement an efficient reduction which appears in the evaluation of the next time step (`compute_deltat()`). The problem of the reduction, while not fully parallel, is well described in the literature which offers us easy to use implementations.

When all those transformations were achieved, we were able to run an implementation taking advantage of the compute power of a GPU.

4.2.4 Lessons learned

- Use INTENT in a FORTRAN code to ease a GPU port.
- Make use of a clear naming convention to help focusing on the different address spaces.

- Simplify algorithm to maximize parallelism.
- Compute instead of store intermediate values.
- Use a simple indexing pattern (preferably ID) whenever possible.
- Minimize data movements by a careful analysis of the array usage.

4.3 OpenCL

4.3.1 Implementation

Having done the CUDA port, moving to other implementations was a straightforward process. All the analysis was done, the constraints understood. It was just a problem of moving the code to yet another language with its idiosyncrasies.

The case of OpenCL can be decomposed into two sub problems: porting the kernel to the OpenCL language and calling the kernels plus operate the data movements.

Moving a kernel from CUDA to OpenCL can almost be done automatically. With a set of clever macros and helper functions, one can easily produce a single source code which can be used either with CUDA or OpenCL. It took us a couple of hours to convert all the kernels to pure OpenCL.

The real difficulty is to call the kernels. While CUDA has been nicely integrated into the C language, OpenCL relies on an API which is rather verbose and cumbersome to use. To ease the burden of OpenCL programming, we developed a set of comfort functions allowing for an almost CUDA looking implementation (the `ocltools.h` and `ocltools.c` files).

```
OCLSETARG12(ker[Loop1KcuTrace], q, dq, c, qxm, qxp, dtdx, Hnxyt, ijmin,
ijmax, zeror, zerol, project);

oclLaunchKernel(ker[Loop1KcuTrace], cqueue, ((ijmax - 1) - (ijmin + 1)),
THREADSSZ);
```

Table 16. OCLxx pseudo library. `OCLSETARGxx` is a macro used to pass `xx` arguments to the kernel (the first parameter).

Using the OCLxx macros and helper function allows us to produce a code as compact as the CUDA version while hiding the complexity of OpenCL. Here we illustrate setting the arguments of the kernel and the compact way to launch it for a given problem size $((ijmax - 1) - (ijmin + 1))$ for a given number of threads (THREADSSZ).

Moving to OpenCL proved itself to be a rather easy task. Most of the troubles we faced were due to immature implementations, OpenCL being a rather fragile environment for most vendors.

4.3.2 Lessons learned

- Helper functions hide unnecessary complexity.
- OpenCL is very close to CUDA and brings portability but not yet performance.

4.4 HMPP

4.4.1 Implementation

The previous remark for OpenCL still holds for HMPP (see [2]). Here the task was lighter since we have to use only the C89 source as a starting point. To transform routines in “codelets” (the HMPP notion of a kernel), we only have to add pragma directives.

```
#pragma hmpp <HGgodunov> Hslope codelet, &
#pragma hmpp <HGgodunov> args[0-1].size = {(Hnxyt + 2) * Hnvar}, &
#pragma hmpp <HGgodunov> args[dq].io = inout, &
#pragma hmpp <HGgodunov> args[2-5].const=true
void
slope(double *RESTRICT q, double *RESTRICT dq, int n,
      const int Hnvar, const int Hnxyt, const double Hslope_type)
{
// ...
#define IHVW(i, v) ((i) + (v) * Hnxyt)
#pragma hmppcg gridify(nbv, i)
    for (nbv = 0; nbv < Hnvar; nbv++) {
        for (i = ijmin + 1; i < ijmax - 1; i++) {
            //...
        }
    }
}
// slope
```

Table 17. HMPP version of the slope() routine

The ONLY modification to the source was to introduce the #pragma directives.

At this stage only a couple of hours were needed to move from a CPU version to a full functioning GPU enabled code. But performances are not there yet. We need to use the data analysis done for CUDA and/or OpenCL to inform HMPP about what needs to be transferred or not and at which point. This is done through the usage of `advancedload` and `noupdate=true` directives as illustrated below. This phase is the most time consuming one because it should be done incrementally to verify that the code is not broken doing so.

The following piece of code illustrates the final stage where every variable has been moved to the GPU in advance of the computation and where every routine operates only on GPU arrays.

```

Void hydro_godunov(int idimStart, double dt, const hydroparam_t H,
                  hydrovar_t * Hv, hydrowork_t * Hw,
                  hydrovarwork_t * Hvw)
{ // ...
  for (idimIndex = 0; idimIndex < 2; idimIndex++) {
    int idim = (idimStart-1 + idimIndex) % 2 + 1;
    if (!H.nstep && idim == 1) {
#pragma hmpp <HGgodunov> allocate
#pragma hmpp <HGgodunov> Hriemann advancedload, args[4-10] /* Buffers and
constant Values */
#pragma          hmpp          <HGgodunov>          Hqlefttright          advancedload,
args[Hnx;Hny;Hnxyt;Hnvar] /* Constant Scalars */
#pragma hmpp <HGgodunov> Htrace advancedload, args[Hscheme;n;dtdx]
#pragma hmpp <HGgodunov> Hslope advancedload, args[Hslope_type;n]
#pragma          hmpp          <HGgodunov>          HequationOfState          advancedload,
args[imin;imax;Hsmallc;Hgamma]
#pragma hmpp <HGgodunov> Hconstoprism advancedload, args[n;Hsmallr]
#pragma hmpp <HGgodunov> Hcmpflx advancedload, args[narray;Hgamma]
#pragma hmpp <HGgodunov> HgatherConservativeVars advancedload, args[4-11]
#pragma hmpp <HGgodunov> HupdateConservativeVars advancedload, args[6-13]
#pragma hmpp <HGgodunov> HgatherConservativeVars advancedload, args[uold]
    }
#pragma hmpp <HGgodunov> Hqlefttright advancedload, args[idim]
#pragma hmpp <HGgodunov> Htrace advancedload, args[dtdx]
    for (j = Hmin; j < Hmax; j++) {
#pragma          hmpp          <HGgodunov>          HgatherConservativeVars          callsite,          args[4-
11].nouupdate=true,
args[u;uold].nouupdate=true,
args[idim].advancedload=true
      gatherConservativeVars(idim, j, uold, u, H.imin, H.imax, H.jmin,
                           H.jmax, H.nvar, H.nxt, H.nyt, H.nxyt);
#pragma hmpp <HGgodunov> HDmemset callsite, args[0-2].nouupdate=true
      Dmemset(dq, 0, (H.nxyt+2) * H.nvar);
      // Convert to primitive variables
#pragma hmpp <HGgodunov> Hconstoprism callsite, args[3-6].nouupdate=true,
args[e;q;u].nouupdate=true
      constoprism(u, q, e, Hdimsize, H.nxyt, H.nvar, H.smallr);
    }
  }
}

```

Table 18. HMPP main driver

This code is an extract of the main driver of the code using HMPP directives to pilot efficiently data movements. We first allocate the variables on the GPU then compute using them, stating that they are already available on the GPU thus avoiding unnecessary data transfers.

The code could be further improved by moving the “`#pragma hmpp <HGgodunov> allocate`” outside of the `hydro_godunov()` and put the full main loop in the `hydro_godunov.c` file. It would also help to have `compute_deltat()` ported to HMPP which hasn't been done in this prototype (it is left as an exercise to the reader).

4.4.2 *Lessons learned*

- The reorganization and analysis done for CUDA/OpenCL is exactly the same for a HMPP port (and potentially for OpenACC).
- Iterate carefully on the data movement optimization and accept performance losses until all unused transfers are removed.

4.5 C99

4.5.1 Implementation

The only difference between the C89 and C99 which is of interest for us is its ability to handle nicely multidimensional arrays. With some word processing we were to have a real C99 implementation. Doing so we lost the FORTRAN style indexing having to permute indexes ($q(\text{Hnxyt}, \text{Hnvar})$ becomes $q[\text{Hnvar}][\text{Hnxyt}]$).

It turned out that, while interesting by itself, this version is useless for further evolutions since C++ is not compatible with this syntax, our goal being producing a full C++ implementation!

```

Void slope(const int n, const int Hnvar, const int Hnxyt,
           const double Hslope_type, double q[Hnvar][Hnxyt],
           double dq[Hnvar][Hnxyt]) {
    int nbv, i, ijmin, ijmax;
// ...
    for (nbv = 0; nbv < Hnvar; nbv++) {
        for (i = ijmin + 1; i < ijmax - 1; i++) {
            // ...
            dq[nbv][i] = dsgn * (double) MIN(dlim, DABS(dcen));
        }
    }
}
// slope

```

Table 19. Our slope routine written using C99

The real advantage of this version is that, while sticking to C, it is much more readable than the C89 version. The impact of the new C11 standard has still to be evaluated in this context.

4.5.2 Lessons learned

- If C is the only language possible, prefer C99 to C89.

4.6 C99 MPI 2D

The C99 version was the opportunity to introduce two new functionalities: a domain decomposition using MPI and a 2D sweep of the domain. This work was done so that it would be easy to back-port it to CUDA and/or other implementations.

4.6.1 MPI implementation

The only impact of the MPI port was to adapt the boundary conditions processed in routine `make_boundary()`. This is a fairly straightforward change done using `Isend` and `Irecv`. Instead of using the MPI functions to describe the boundaries layout in memory (as it is done in the FORTRAN code), we implemented our own packing routines so that they could be easily adapted to a GPU kernel since data movement from/to the GPU are done only on continuous regions of memory.

We also chose to implement a k-D tree decomposition instead of a simpler one as used in the FORTRAN versions. This has some impacts: the code can only use a power of 2 processors and the topology of the MPI processes doesn't match easily to the domain numbering. This might produce artefacts performance wise (not measured as of today). A more regular decomposition is an option for a future version. If introduced as an option, this second scheme could be compared to the k-D tree to see the influence of the underlying network and task placement.

5	7	13	15
4	6	12	14
1	3	9	11
0	2	8	10

Figure 18. A k-D tree decomposition for 16 MPI process.

4.6.2 2D sweep implementation

The original algorithm uses the well known alternate directions scheme. This algorithm processes each line of the domain in a first phase then all the columns in a second phase. The processing phase deals with each X of the line (resp. each Y of the column). Therefore we have a loop over Y which includes a loop over X (resp. a loop over X including a loop over Y).

A simple transformation could be to enlarge the computed domain. Now, we process a band of line (resp. a band of column) instead of a single line (resp. column). This transformation will increase the number of elements processed during each iteration of the external loop and decrease the number of iterations needed to process the whole domain. With increasing cache sizes, this could yield to a faster code.

The code has now a parameter telling the size of the band to use. -1 means use the full domain height (resp. width).

4.6.3 Lessons learned

- Increasing the workload of a computational subroutine will increase the GPU saturation.

4.7 *CUDA MPI 2D*

4.7.1 *Implementation*

During all the experiments done with the first CUDA implementation, we noticed that the speedup was increasing with the size of the processed domain. To be of interest, the CUDA version had to process a very large domain both in X and Y. This was due to the fact that the kernels were operating only on a single line (resp. column). Unless we're computing a very large domain (above 5000x5000), there is no chance to saturate the hardware with a single line (or column) of elements. Using the 2D sweep implemented for the C99 version, back-ported to CUDA, allowed us to increase dramatically the load on the GPU at the expense of local memory.

Smaller domains fitting in the GPU memory can now be processed in a single step for both directions. For larger domains, the proper bandwidth has to be evaluated. This evaluation has not been automated yet. Using this feature, we can measure effective speedups sooner than what was available with the initial version.

Finally, the coupling with MPI reuses what has been implemented for the C99 version. Since we have the packing routines which can easily be transformed into kernels, the exchange of boundaries from one GPU to the other has one additional step compared to the CPU version: copy the result of the packing (resp. unpacking) kernel from the GPU to the CPU on sending side (resp. from the CPU to the GPU on the receive side). Otherwise, the code remains unchanged.

4.7.2 *Lessons learned*

- A GPU shows potential only if saturated. Favor a limited number of kernels working on larger datasets rather than iterate on kernel operating on smaller domains.
- Pack/unpack MPI buffers manually so that it can be easily ported to a GPU.

4.8 UPC version

4.8.1 Language presentation

UPC (Unified Parallel C [9,10]) is an extension of ISO C that provides access to a shared partitioned address space, where variables may be directly used by any processor, but each variable is associated with a single thread. The number of threads is fixed during computation.

UPC adds a new type qualifier (`shared`) in order to specify that a variable is shared between all the others UPC threads, without this qualifier variables keep private status.

```
int result;
static shared int counter;
```

Table 20. The shared keyword

Shared variables cannot have automatic storage duration, because remote access of a variable would not be possible. The shared address space is logically partitioned, and each shared variable has affinity to only one of the threads. This enables programmers to co-locate data and processing, thus exploiting data locality for improved performance.

```
static shared int array[300];
```

Table 21. shared array

UPC identifier `THREADS` give the total number of threads, `MYTHREAD` give the thread number (value between 0 and `THREADS-1`).

By default, the partitioning uses the round-robin algorithm, `array[0]` has affinity to thread 0, `array[1]` has affinity to thread 1 and so on, after one round we wrap around giving `array[THREADS]` to thread 0, `array[THREADS+1]` to thread 1, and so on. Since in many cases, this default behavior is not optimal, the array distribution can be altered by specifying a block size in the declaration.

```
static shared[BLOCKSIZE] int array[300];
```

Table 22. shared array with block size

So now the elements from `array[0]` to `array[BLOCKSIZE-1]` have affinity to thread 0, the elements from `array[BLOCKSIZE]` to `array[2*BLOCKSIZE-1]` have affinity to thread 1, and so on.

Access to shared variables can be done with pointers, there are four UPC pointer types:

- private pointers pointing to private space,
- private pointers pointing to shared space,
- shared pointers pointing to shared space,
- and shared pointers to private space.

The last one must be avoided because it points on a space only accessible by one thread.

```
int *p; /* private pointers to private space */
shared[BLOCKSIZE] int *p; /* private pointers to shared space */
shared[BLOCKSIZE] int * shared p; /* shared pointers to shared space */
```

Table 23. UPC pointer types

The UPC construct `upc_forall` is a workload sharing iteration statement that looks like the C `for` loop statement.

```
upc_forall (expression1; expression2; expression3; affinity)
```

Table 24. `upc_forall` syntax

The first three arguments of the `upc_forall` statement are similar to that of the C `for` statement (initialization, test and update phase). The fourth argument determines which thread executes a given iteration. The expression must be the same between all threads and can be an integer or a pointer to shared. When affinity is an integer, all iterations for which the value affinity module THREADS matches the thread number, will be executed by that thread. When affinity is a pointer to shared, a thread executes the iteration if the object pointed by it has affinity to that thread.

Dynamic memory allocation of the shared space can be done in UPC. There are three functions. The first one `upc_all_alloc` is a collective function:

```
shared void *upc_all_alloc(size_t nblocks, size_t nbytes);
```

Table 25. `upc_alloc` syntax

It's similar to a static declaration:

```
shared[nbytes] char[nblocks*nbytes];
```

Table 26. static declaration

`upc_global_alloc` is similar to `upc_all_alloc` except that it is not a collective function. If called by multiple threads, all threads that make the call get different allocations.

```
shared void * upc_global_alloc(size_t nblocks, size_t nbytes);
```

Table 27. `upc_global_alloc` syntax

And `upc_alloc` allocates a shared storage with affinity to the calling thread.

```
shared void * upc_alloc(size_t nbytes);
```

Table 28. `upc_alloc` syntax

`upc_free` can be used to freeing such allocated spaces.

4.8.2 Heat conduction equation

In order to understand how to do domain decomposition in HYDRO, we first study a simpler example: a 2D-Stencil code.

4.8.2.1 Algorithm

```
tab[i][j] = ( tab[i-1][j]+tab[i+1][j]+tab[i][j-1]+tab[i][j+1] )/4
```

Table 29. Heat conduction equation algorithm

30	31	32	33	34
25	0	0	0	29
20	0	0	0	24
15	0	0	0	19
10	0	0	0	14
5	0	0	0	9
0	1	2	3	4

Table 30. Domain with boundary values for nx=7, ny=5

The boundary conditions are fixed to the 1D-index: $(j+ny*i)$.

We iterate until the maximum difference between two iterations is less than a fixed value. In the end of all iterations all values must be near their 1D-indexes.

30	31	32	33	34
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

Table 31. Final values in the domain nx=7, ny=5

We use a 3D array, the first dimension provides a way to keep values at iteration t and iteration t-1, the second dimension has a size of nx*ny The kernel of computation is:

```
for (itercol=1; itercol<nx-1; itercol++) {
    for (iterrow=1; iterrow<ny-1; iterrow++) {
        int offset = itercol*ny+iterrow;
        domain[t][offset] =
            (domain[tmun][offset-ny] +
             domain[tmun][offset+ny] +
             domain[tmun][offset-1] +
             domain[tmun][offset+1] )/4;
        diff = fabs(domain[t][offset] -
                    domain[tmun][offset]);
        diffmax = MAX(diffmax,diff); } }
```

Table 32. Heat equation main code

4.8.2.2 First implementation with BLOCKSIZE

The main trouble with UPC is that BLOCKSIZE is required to be known at compile time. So we have to use a round-robin distribution that has poor performance using less effectively the cache system. Another drawback to distributions in UPC is that they are one-dimensional. A 2D-block distribution is not possible.

In the first method, we accept these constraints and use a BLOCKSIZE fixed during compilation time.

```

#define BLOCKSIZE 30
shared[BLOCKSIZE] double** domain;
domain=(shared[BLOCKSIZE] double **)
    malloc(2*sizeof(shared[BLOCKSIZE] double*));
domain[0] = upc_all_alloc(1+(nx*ny)/BLOCKSIZE,BLOCKSIZE*sizeof(double));
domain[1] = upc_all_alloc(1+(nx*ny)/BLOCKSIZE,BLOCKSIZE*sizeof(double));

```

Table 33. Domain construction

```

for(itercol=1; itercol<nx-1; itercol++) {
    upc_forall(iterrow=1;
        iterrow<ny-1;
        iterrow++;
        &(domain[0][itercol*ny+iterrow])) {
        domain[t][itercol*ny+iterrow] =
            ( domain[tmun][(itercol-1)*ny+iterrow] +
              domain[tmun][(itercol+1)*ny+iterrow] +
              domain[tmun][itercol*ny+iterrow-1] +
              domain[tmun][itercol*ny+iterrow+1] )/4;
        diff = fabs(domain[t][itercol*ny+iterrow] -
                    domain[tmun][itercol*ny+iterrow]);
        ldifffmax = MAX(ldifffmax,diff); } }

```

Table 34. Main loop

The interest of this version is to offer few changes to the code, but with a poor distribution in terms of performance (usage of cache).

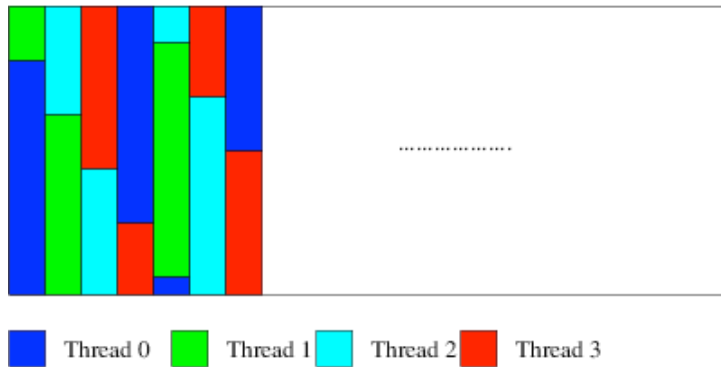


Figure 19. Domain distribution on 4 threads

4.8.2.3 Second implementation

If we want a more appropriate distribution, we will allocate only one block on each thread. We compute the blocksize ($l_{nx} \times l_{ny}$) from the problem size ($n_x \times n_y$). As each block has the same local size, there could be a risk that the last thread has less elements due to rounding.

```
domain = (shared double **) malloc(2*sizeof(shared double*));
domain[0] = upc_all_alloc(THREADS, lnx*ny*sizeof(double));
domain[1] = upc_all_alloc(THREADS, lnx*ny*sizeof(double));
```

Table 35. Domain construction

It looks like the following declaration:

```
shared[lnx*ny] double domain[2][lnx*ny*THREADS];
```

Table 36. Domain construction

The trouble here is that the blocksize qualifier of a pointer is a part of the type. l_{nx} and l_{ny} must be constants known at compile time. So we cannot use `domain` as `shared[lnx*ny] double**` object type, therefore we cannot longer use UPC pointer arithmetic. We have to access elements by calling a function, which computes the thread affinity and the offset.

Here is the function `getValue` which returns a pointer on `tab2d[i][j]` taking into account the blocksize `bx`.

```
shared[] double* getValue(shared double* tab2d,
                          size_t i, size_t j,
                          size_t bx, size_t ny) {
    int thrd, offset;
    thrd = i/bx;
    offset = j+ny*(i%bx);
    return &((shared[] double *) (tab2d+thrd))[offset];
}
```

Table 37. function `getValue`

Unfortunately the kernel becomes a lot less readable:

```
upc_forall(itercol=1; itercol<nx-1; itercol++; itercol%lnx) {
    for (iterrow=1; iterrow<ny-1; iterrow++) {
        *getValue(domain[t],itercol,iterrow,lnx,ny) =
            ( *getValue(domain[tmun],itercol-1,iterrow,lnx,ny) +
              *getValue(domain[tmun],itercol+1,iterrow,lnx,ny) +
              *getValue(domain[tmun],itercol,iterrow-1,lnx,ny) +
              *getValue(domain[tmun],itercol,iterrow+1,lnx,ny) )/4;
        diff = fabs(*getValue(domain[t],itercol,iterrow,lnx,ny) -
                    *getValue(domain[tmun],itercol,iterrow,lnx,ny));
        ldiffmax = MAX(ldiffmax,diff);
    }
}
```

Table 38. Main loop with new declaration

4.8.2.4 Third implementation

If we want a more balanced distribution, we have to imitate what we are doing with MPI: each thread allocates his domain in the shared area and writes the location in a shared array of pointer. Each thread computes the local bound x in $[l_{dx}, l_{fx}]$, and allocates the local domain:

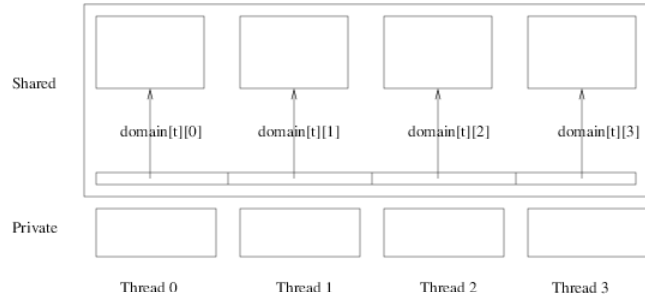


Figure 20. Domain distribution on 4 threads

```
shared[] double* shared domain[2][THREADS];
domain[0][MYTHREAD] = upc_alloc( (lfx-lfx)*ny*sizeof(double));
domain[1][MYTHREAD] = upc_alloc( (lfx-lfx)*ny*sizeof(double));
```

Table 39. Domain construction

With this structure we have the same problem as mentioned in section 0: in order to access an element we must use a function getValue:

```
shared[] double* getValue(shared[] double* shared tab2d[THREADS],
                          size_t i, size_t j, size_t nx, size_t ny) {
    int thrd, offset;
    thrd = ((i+1)*THREADS-1)/nx;
    offset = j+ny*(i - ((int) (thrd*(double) nx))/THREADS);
    return &(tab2d[thrd][offset]);
}
```

Table 40. addressing function getValue

The computation kernel is the same as described in Table 38.

4.8.2.5 Cost of pointer to shared

The three methods have poor performance in comparison with the C version, this is mainly due to the use of a pointer to shared variable instead of the C pointer arithmetic. On our Power 7 P755 with Berkeley UPC the use of pointer-to-shared is twenty times slower than the use of a C regular pointer.

There is a way to cast a pointer-to-share in a regular pointer: when the thread has an affinity with the pointed area, we can then use pointer arithmetic but we must do bound checking. In fact in this case we have to do something that looks like what we do with MPI, separate the local domain into an inner and a boundary domain and do the computation with C regular pointer in the inner domain. Unlike MPI, UPC doesn't have high level data layout: MPI Datatype, MPI Communicators to group the threads ...

4.8.3 Hydro UPC

For the UPC version of HYDRO, we want to do the least possible changes to the code, so we use the first method described in section 4.8.2.2.

The modifications are in four areas, the declaration of the domain, the initialization of the domain, the computation of the delta time and the Godunov function.

4.8.3.1 Declaration of the domain

The local domain is in the structure `hydrovar_t`:

```
// Hydrovar holds the whole 2D problem for all variables
typedef struct _hydrovar {
    double *uold;           // nxt, nyt, nvar allocated as (nxt * nyt), nvar
} hydrovar_t;             // 1:nvar
```

Table 41. Structure `hydrovar_t` in C

For the UPC version, we fixed the `BLOCKSIZE` value to 1024 and added a shared array for the computation of delta time (`dt`).

```
// Hydrovar holds the whole 2D problem for all variables
#define BLOCKSIZE 1024
typedef struct _hydrovar {
// nxt, nyt, nvar allocated as (nxt * nyt), nvar
    shared[BLOCKSIZE] double *uold;
    shared double * dt;
} hydrovar_t;             // 1:nvar
```

Table 42. Structure `hydrovar_t` in UPC

4.8.3.2 Initialization of the domain

So the allocation of the domain changes a little:

```
// allocate uold for each conservative variable
Hv->uold = (double *) calloc(H->nvar * H->nxt * H->nyt, sizeof(double));
```

Table 43. Dynamic allocation of the domain in C

becomes

```
// allocate uold for each conservative variable
sizetot = H->nvar * H->nxt * H->nyt;
nbblock = ((sizetot % BLOCKSIZE) == 0) ?
    sizetot/BLOCKSIZE : 1+sizetot/BLOCKSIZE;
Hv->uold = (shared[BLOCKSIZE] double *)
    upc_all_alloc(nbblock, BLOCKSIZE*sizeof(double));
Hv->dt = (shared double *) upc_all_alloc(THREADS, sizeof(shared double));
```

Table 44. Dynamic allocation of the domain in UPC

4.8.3.3 Computation of delta time

The original version is here:

```

for (j = H.jmin + ExtraLayer; j < H.jmax - ExtraLayer; j++) {
    ComputeQEforRow(j, Hv->uold, Hv->q, Hw->e, H.smallr, H.nx, H.nxt,
                    H.nyt, H.nxyt);
    equation_of_state(&Hv->q[IHvw(0, ID)], Hw->e,
                    &Hv->q[IHvw(0, IP)], Hw->c, 0, H.nx, H.smallc,
                    H.gamma);
    courantOnXY(&cournox, &cournoy, H.nx, H.nxyt, Hw->c, Hv->q);
}
Free(Hv->q);
Free(Hw->e);
Free(Hw->c);
*dt = H.courant_factor * H.dx / MAX(cournox, MAX(cournoy, H.smallc));

```

Table 45. Compute_deltat function in C

The UPC version is:

```

upc_forall (j = H.jmin + ExtraLayer; j < H.jmax - ExtraLayer; j++; j) {
    ComputeQEforRow(j, Hv->uold, Hv->q, Hw->e,
                    H.smallr, H.nx, H.nxt, H.nyt, H.nxyt);
    equation_of_state(&Hv->q[IHvw(0, ID)], Hw->e,
                    &Hv->q[IHvw(0, IP)], Hw->c, 0, H.nx, H.smallc,
                    H.gamma);
    courantOnXY(&cournox, &cournoy, H.nx, H.nxyt, Hw->c, Hv->q);
}
Free(Hv->q);
Free(Hw->e);
Free(Hw->c);
// Reduce min sur dt
Hv->dt[MYTHREAD] = H.courant_factor * H.dx
                  / MAX(cournox, MAX(cournoy, H.smallc));
upc all reduceD(dt, Hv->dt, UPC MIN, THREADS, 1, NULL, 0);

```

Table 46. Compute_deltat function in UPC

Since using affinity for distribution of iterations needs a lot of change, we use the thread number for distribution. We lack here the load balancing facilities of OpenMP (SCHEDULE)

4.8.3.4 Godunov function

The boundary conditions update (make_boundary.c) is done with taking into account the affinity of the loops like this:

```

for (j = H.jmin + ExtraLayer; j < H.jmax - ExtraLayer; j++) {
    Hv->uold[IHv(i, j, ivar)] = Hv->uold[IHv(i0, j, ivar)] * sign;
...

```

Table 47. make_boundary C function

becomes:

```

upc_forall (j = H.jmin + ExtraLayer;
           j < H.jmax - ExtraLayer;
           j++;
           &(Hv->uold[IHv(i, j, ivar)])) {
    Hv->uold[IHv(i, j, ivar)] = Hv->uold[IHv(i0, j, ivar)] * sign;
...

```

Table 48. make_boundary UPC function

In Godunov computation, like in the computation of delta time we use the thread number for distribution of the lines like:

```

if (idim == 1) {
    for (j = H.jmin + ExtraLayer; j < H.jmax - ExtraLayer; j++) {
...

```

Table 49. hydro_godunov C function

becomes:

```

if (idim == 1) {
    upc_forall(j = H.jmin + ExtraLayer; j < H.jmax - ExtraLayer; j++; j) {
...

```

Table 50. hydro_godunov UPC function

4.8.3.5 Comments on the changes made in the code

The modifications are light, around 50 lines modified, mainly because of adding the shared attribute in declaration and definition functions, 20 lines are added to the major part in the initialization phase. We spent roughly 15 days experimenting with the different options to do the domain decomposition in order to evaluate benefits and drawbacks of each.

4.8.3.6 Performance

The characteristics of used data sets:

- Number of points of the domain: $N_x \times N_y = 10000^2 = 1 \times 10^8$,
- Number of iterations: 10
- Total memory used: 3Gb

We ran the test cases on two platforms. The characteristics of the target machines are:

- an IBM P755 (IDRIS, France), 32 cores per node, 128 Gb/node, Berkeley UPC compiler,
- the Tier-0 system CRAY XE6 HERMIT (HLRS, Germany), 16 cores per node, 32Gb/node, Cray UPC compiler.

	C	UPC					
	Mono	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Elapsed time (s)	1112	1722	868	440	321	217	131
Speedup	1	0.7	1.3	2.5	3.5	5.1	8.5

Table 51. UPC HYDRO performance on IBM P77

	C	UPC					
	Mono	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Elapsed time (s)	2287	2669	1948	1140	644	360	200
Speedup	1	0.9	1.2	2	3.6	6.4	11.4

Table 52. UPC HYDRO performance on CRAY XE6

In the Godunov computation, HYDRO uses a working buffer for reading values in the shared domain, so all computation are not done with pointer-to-shared. This explains that the sequential UPC version has poorer performance than the C sequential version. As mentioned for the heat conduction equation, the use of pointer-to-shared is twenty times slower than the use of a C regular pointer (see 4.8.2.5).

4.8.3.7 Lessons learned

UPC provides a way to quickly add a global view of memory. It simplifies writing a parallel shared memory version of an ordinary C code. But in order to achieve performance, there are some disadvantages:

- `BLOCKSIZE` is required to be known at compile time, making it hard to have a better cache use.
- The distribution is one dimensional making a 2D-block distribution impossible without using a shared array of pointers to local arrays.
- There is no equivalent to MPI datatype, communicators or OpenMP Schedule constructions.
- In order to achieve performance, we sometimes have to use the same decomposition that we develop in MPI (halo).
- The compilers are currently in experimental phase (at very variable levels of maturity).
- The UPC runtime needs also to mature to offer faster than MPI performance (using efficient one-sided communications).
- Data race must be avoided using synchronization (barrier, locks, ...).

4.9 The VTK output

To make a film of the results, we need to have an output file which can be interpreted by a visualization software. We chose paraview, an open-source software, which can read files from a parallel simulations and which can also run in parallel.

The tree structure of the post-processing files is as follows:

Root level	Level 1	Level 2	Level 3	Comment
Hydro. pvd				The VTK description of this simulation pointing to all the .pvtr files
Dep/				The main container for the post-processing files
	outnum/100			The upper part of the output number
		outnum % 100/		The 2 lower digit of the output number
			Hydro_rank_outnum. vtr	One file per MPI process
			Hydro_outnum. pvtr	One VTK description of this output in time, pointing to all the .vtr files of this directory

Table 53. Directory structure of the post-processing files

We made this structure for the following reasons:

1. To be able to visualize only one output time by loading only a single .pvtr file.
2. To have the means of producing a film from all output dumps by loading the .pvd file.
3. To have the capacity to saturate the file system by writing many (one per process) files simultaneously. This is especially important for Prace prototype owners who want to exercise the I/O part of their machine with a load that mimics real codes.
4. To be able to test the file system in read mode through a parallel software.
5. To create a full tree to test the metadata part of the file system.

Since all VTK processing is located in a single source file, it is therefore easy to adapt the structure of the output tree if one wants to test the behavior of a file system having many entries (i.e. all .vtr files in the same directory).

4.10 Test cases

4.10.1 Centred explosion

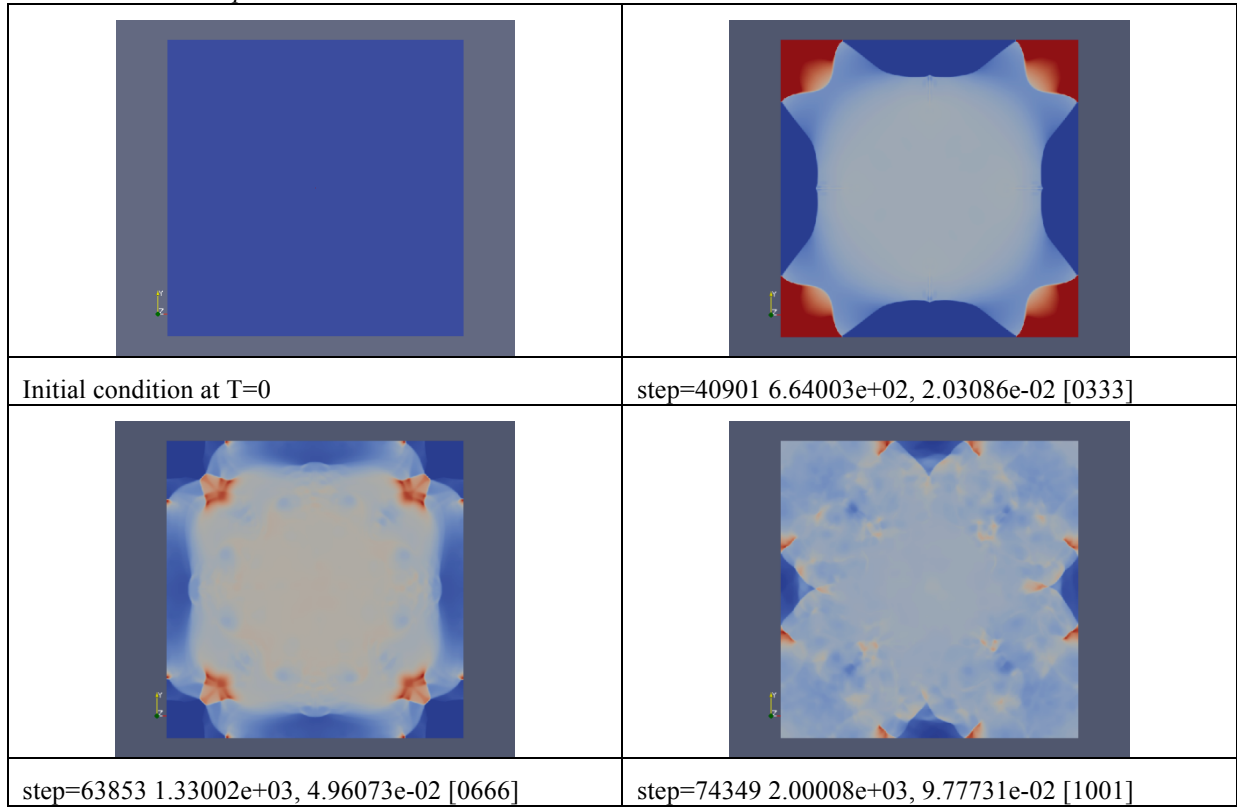


Figure 21 : Centred explosion at different time.

These pictures show the evolution of the simulation for a 600x600 domain. The initial point is a tiny red dot on the first image (not visible). The iteration number, the physical time, the time step and the number of the frame are reported. They should not vary with the port or optimizations of the program.

4.10.2 Timing Results

To compare the different versions, we use a 9000x9000 domain with no VTK output to avoid the I/O time which introduces timings variations according to the load of the machine.

We compute 10 iterations and report the total elapsed time for a single CPU or a single GPU.

9000x9000	C99 2D B=1	C99 2D B=20	CUDA 2D B=1	CUDA 2D B=20	HMPP CUDA	OpenCL
Westmere-EP	701.724s	729.978s	N/A	N/A	N/A	N/A
C2050	N/A	N/A	117.936s	55.150s	175.746s	269.865s
Speedup	1	0.96	5.95	12.72	3.99	2.60

Table 54. Comparison of the different implementations

B=1 means 1 line (column) at a time, B=20 means a band of 20 lines (columns) at a time. B=20 was chosen because at higher values the speedup wasn't much improved. On the other hand, it shows that 2D coding could have a negative effect on the CPU version (here true blocking should be experimented).

If we select a 900x900 domain, it shows that loading the GPU is important to amortize the cost of data transfer on the PCI-Express.

900x900	C99 2D B=1	C99 2D B=20	CUDA 2D B=1	CUDA 2D B=900
Westmere-EP	5.907s	5.951s	N/A	N/A
C2050	N/A	N/A	7.332s	1.066s
Speedup	1	0.99	0.81	5.54

Table 55. Influence of the load on the GPU

This small example illustrates that codes should be adapted to cope with small problems sizes if one wants to have performance improvements for all cases.

5 Conclusions

This document is a description of the work achieved on the HYDRO code. This code includes classical algorithms we can find in real production code. It is neither a big software nor a small kernel. That's why we have explored and implemented in a reasonable time new high performance programming techniques to parallelize the code. We learned many lessons concerning:

- the way to implement the different parallelization techniques,
- the maturity of the compilers or the libraries we used,
- the performance we can obtain, the scalability of the code.

These lessons can be very fruitful for programmers who have to parallelize high performance applications and we think they have the potential to facilitate significant improvements in real applications performance. Many more versions should come in the future.

6 Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources CURIE at TGCC (France), JUGENE at Juelich (Germany), HERMIT at HLRS (Germany), the PRACE PP-WP8 prototype at CEA/DAM and the OpenGPU [4] prototype located at CEA/DAM (France) which has been made available to the PRACE-IIP-WP9 community.

7 References

1. More details on RAMSES can be obtained on the following web sites :
http://irfu.cea.fr/Sap/en/Phoea/Vie_des_labos/Ast/ast_visu.php?id_ast=904
<http://web.me.com/romain.teyssier/Site/RAMSES.html>
2. Godunov, S. K. (1959), "A Difference Scheme for Numerical Solution of Discontinuous Solution of Hydrodynamic Equations", *Math. Sbornik*, **47**, 271–306, translated US Joint Publ. Res. Service, JPRS 7226, 1969.
3. Roe, P. L. (1981), "Approximate Riemann solvers, parameter vectors and difference schemes", *J. Comput. Phys.* **43** (2): 357–372, Bibcode 1981JCoPh..43..357R, doi:10.1016/0021-9991(81)90128-5
4. MPI - The Message Passing Interface standard, <http://www.mcs.anl.gov/research/projects/mpi/>
5. OpenMP - Open Multi-Processing specifications, <http://openmp.org/wp/openmp-specifications/>
6. HMPP is a many-core environment from CAPS: <http://www.caps-entreprise.com>
7. Hybrid MPI-OpenMP Programming, P.-Fr. Lavallée and Ph. Wautelet, http://www.idris.fr/data/cours/hybride/choix_doc.html
8. OpenGPU is a project of the Systematic cluster of competitiveness
9. W. Carlson et. al., *UPC: Distributed Shared Memory Programming*, Book of Wiley Inter-Science (2005)
10. Berkeley UPC, <http://upc.lbl.gov>