In [1]:
```
import matplotlib.pyplot as plt
#For latex documents
#textwidth = 13
textwidth=15
figheight = 0.25*textwidth

plt.rc('figure', figsize=(0.66*textwidth,figheight))
#plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rc('grid', linestyle="--")
plt.rc('grid', alpha="0.5")
plt.rc('axes', grid=True)

def latex_float(f):
    float_str = "{0:.2g}".format(f)
    if "e" in float_str:
        base, exponent = float_str.split("e")
        if(base == "1"):
            return r"10^{{{1}}}".format(base, int(exponent))
        return r"{0} \times 10^{{{1}}}".format(base, int(exponent))
    else:
        return float_str

import numpy as np
import re
from matplotlib.ticker import FormatStrFormatter

!mkdir fig
```

```
mkdir: cannot create directory ‘fig’: File exists
```

Cas test

le run.def est le suivant, est remplacé par le nombre de threads sur la verticale:

- 1 pour mpmd
- Le nombre de threads annoncé pour scal

In [2]:
```
!cat scripts/run_scal.def
```

```
INCLUDEDEF = const.def

#---------------- Mesh ----------------

# Number of subdivisions on a main triangle : integer (default=40)
nbp = 41

# Number of vertical layers : integer (default=19)
llm = 79

# Vertical grid : [std|ncar|ncarl30] (default=std)
disvert = std

# Mesh optimisation : number of iterations : integer (default=0)
optim_it = 100

# Sub splitting of main rhombus : integer (default=1)
nsplit_i = 2
nsplit_j = 2
omp_level_size=<thread_h>

#---------------- Numerics ----------------

# Time step in s : real (default=480)
dt = 720

# Dissipation time for grad(div) : real (default=5000)
tau_graddiv = 9000

# Exponent of grad(div) disspation : integer (default=1)
nitergdiv = 2

# Dissipation time for curl(curl) : real (default=5000)
tau_gradrot = 9000

# Exponent of curl(curl) disspation : integer (default=1)
nitergrot = 2

# Dissipation time for div(grad) : real (default=5000)
tau_divgrad = 9000

# Exponent of div(grad) disspation : integer (default=1)
niterdivgrad = 2


#---------------- Time ----------------

# Run length in s : real (default=??)
run_length = 86400

# Interval in s between two outputs : integer (default=??)
write_period = 86400


#---------------- Physical parameters ----------------

# Number of tracers : integer (default=1)
nqtot = 5

# Initial state :
#   [jablonowsky06|academic|dcmip[1-4]|heldsz|dcmip2_schaer_noshear] (default=j
ablonowsky06)
etat0 = dcmip2016_baroclinic_wave
```

Extraire les données

In [3]:
```
times_mpmd={}
for line in open("data/out_mpmd"):
    ds,nb_proc,time= re.match(".*/(.*)_(\d*).0.1.0:.*:\s*(\d*.\d*)", line).grou
ps()
    times_mpmd[ds,int(nb_proc)]=float(time)

times_scal={}
for line in open("data/out_scal_vec"):
    nb_proc,nb_thread,time= re.match(".*/p(\d*)_th(\d*):.*:\s*(\d*.\d*)", line)
.groups()
    times_scal[int(nb_proc),int(nb_thread)]=float(time)
```

Efficacité MPMD

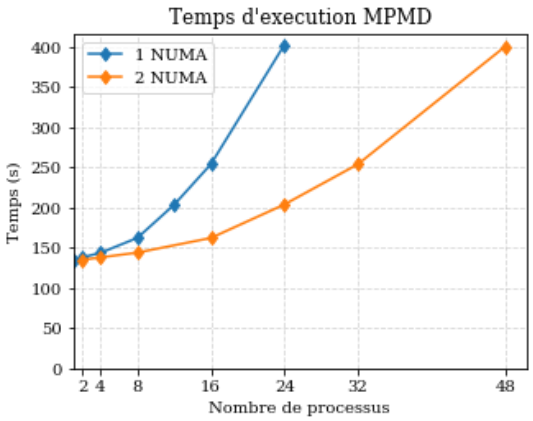
Plusieurs executions indépendantes sont lancées en même temps sur un noeud Joliot-Curie (2x24 proc) :

- single en remplissant un noeud NUMA
- double en remplissant les 2 noeuds NUMA

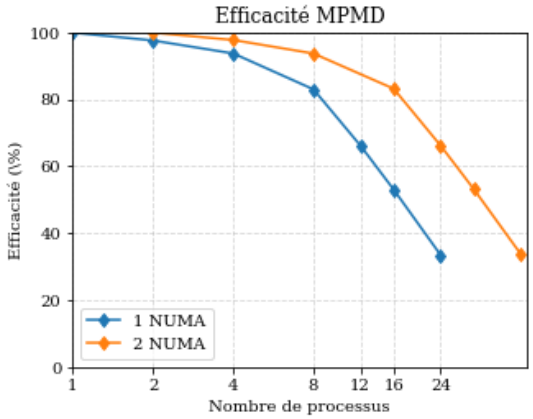
In [4]:
```
#Script d'execution :
!cat scripts/batch_mpmd.sh
#Résultats
#!grep -r "Time elapsed" out/*
print(times_mpmd)
```

```
#!/usr/bin/bash
#SBATCH -J Dynamico_scal_mpmd
#SBATCH -N 1
#SBATCH --exclusive
#SBATCH -A gen0826@skylake
#SBATCH -p skylake
#SBATCH -o bench_mpmd.out
#SBATCH -e bench_mpmd.err

set -x

mkdir out

rm run.def
cp run_mpmd.def run.def

for nb_proc in 1 2 4 8 12 16 24 48
do
    for (( i=0; i<$nb_proc; i++ ))
    do
        OMP_NUM_THREADS=1 srun -n 1 -m cyclic:cyclic -o out/${nb_proc}.$i -slot
-list $i ./icosa_gcm &
        pids[${i}]=$!
    done

    # wait for all pids
    for pid in ${pids[*]}; do
        wait $pid
    done
done

{('double', 1): 135.107, ('double', 12): 204.0884, ('double', 16): 254.2374, ('
double', 2): 138.1456, ('double', 24): 400.6108, ('double', 4): 144.1014, ('dou
ble', 8): 162.3409, ('single', 1): 134.921, ('single', 12): 203.7471, ('single'
, 16): 254.3122, ('single', 2): 138.167, ('single', 24): 403.0065, ('single', 4
): 143.8789, ('single', 8): 162.4821}
```

In [5]:
```
def plot_times_mpmd() :
    plt.figure(figsize=(0.33*textwidth, figheight))
    procs = np.array(sorted(set([p for (s,p) in times_mpmd.keys()])))
    times = np.array([times_mpmd["single",p] for p in procs])
    plt.plot(procs, times, 'd-', label="1 NUMA")
    times = np.array([times_mpmd["double",p] for p in procs])
    plt.plot(2*procs, times, 'd-', label="2 NUMA")
    plt.xlim(1)
    plt.ylim(0)
    plt.xticks(2*procs)
    plt.title("Temps d'execution MPMD")
    plt.xlabel("Nombre de processus")
    plt.ylabel("Temps (s)")
    plt.legend()
    plt.savefig("fig/times_mpmd.pdf", transparent=True, bbox_inches='tight')
    plt.show()

plot_times_mpmd()
```



In [6]:
```
def plot_eff_mpmd() :
    plt.figure(figsize=(0.33*textwidth, figheight))
    procs = np.array(sorted(set([p for (s,p) in times_mpmd.keys() if s=="double
"])))
    times = np.array([times_mpmd['single',p] for p in procs])
    plt.plot(procs, times[0]/times*100, '-d', label='1 NUMA')
    times = np.array([times_mpmd['double',p] for p in procs])
    plt.plot(2*procs, times[0]/times*100, '-d', label='2 NUMA')
    plt.ylim([0,100])
    plt.xlim(1)
    plt.xscale('log',basex=2)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%i'))
    plt.xticks(procs)
    plt.title("Efficacité MPMD")
    plt.xlabel("Nombre de processus")
    plt.ylabel("Efficacité (\%)")
    plt.legend()
    plt.savefig("fig/eff_mpmd.pdf", transparent=True, bbox_inches='tight')
    plt.show()

plot_eff_mpmd()
```



Observations

- La performance est moins bonne avec un socket chargé (probablement à cause de la contension mémoire)
- Même performance sur 1 ou 2 sockets (24 ou 48 procs) : confirme l'hypothèse de la contension mémoire

MPI + OpenMP

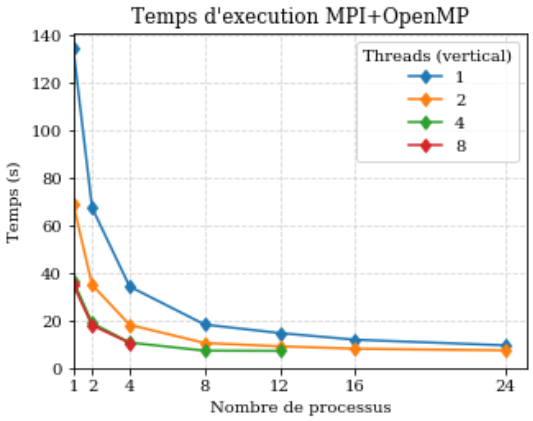
Dynamico est lancé en modifiant le nombre de process et de threads (vertical) sur un noeud d'Irene (2*24 coeurs) compilé avec ou sans AVX (-no-vec -no-simd VS -xHost)

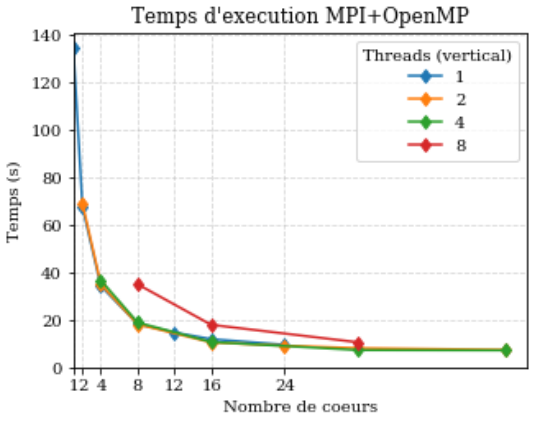
In [7]:
```
#Script d'execution :
!cat scripts/batch_scal.sh
#Résultats
#!grep -r "Time elapsed" out/*
print(times_scal)
```

```
#!/usr/bin/bash
#SBATCH -J Dynamico_scal
#SBATCH -N 1
#SBATCH --exclusive
#SBATCH -A gen0826@skylake
#SBATCH -p skylake
#SBATCH -o out_scal.o
#SBATCH -e out_scal.e

set -x

export OMP_STACKSIZE=500M

mkdir out

for proc in 1 2 4 8 12 16 24 48
do
    for thread_h in 1 2 4 8
    do
        dir=p${proc}_th${thread_h}
        sed 's/<thread_h>/'$thread_h'/g' run_scal.def > run.def
        OMP_NUM_THREADS=$thread_h srun -c $thread_h -n $proc -m cyclic:cyclic .
/icosa_gcm > out/$dir
    done
done
```

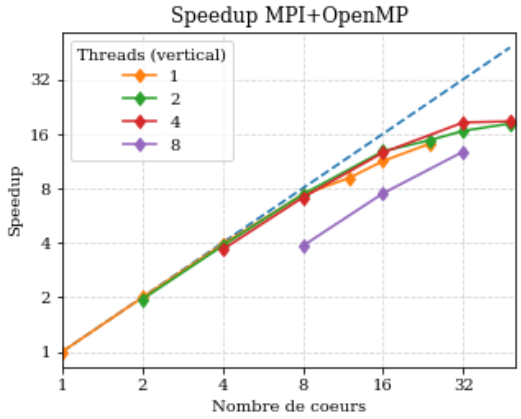
```
{(12, 1): 14.6861, (12, 2): 9.1157, (12, 4): 7.1735, (16, 1): 11.8888, (16, 2):
8.0924, (1, 1): 134.5158, (1, 2): 68.9853, (1, 4): 36.6491, (1, 8): 34.9979, (2
4, 1): 9.5257, (24, 2): 7.3785, (2, 1): 67.4292, (2, 2): 34.9793, (2, 4): 18.89
98, (2, 8): 17.9308, (4, 1): 34.2664, (4, 2): 18.1457, (4, 4): 10.6765, (4, 8):
10.5465, (8, 1): 18.2464, (8, 2): 10.4888, (8, 4): 7.2711}
```

In [8]:
```
def plot_times_parallel(cpu_count, cpu_label) :
    plt.figure(figsize=(0.33*textwidth, figheight))

    threads = np.array(sorted([a for a in set([t for (p,t) in times_scal.keys()
])]))
    for thread in threads:
        procs = np.array(sorted([p for (p,t) in times_scal.keys() if t==thread]
))
        times = np.array([times_scal[p,thread] for p in procs])
        plt.plot(cpu_count(procs,thread), times, 'd-', label=str(thread))
    plt.xlim(1)
    plt.ylim(0)
    plt.xticks(np.array(sorted([p for (p,t) in times_scal.keys() if t==1])))
    plt.title("Temps d'execution MPI+OpenMP")
    plt.xlabel("Nombre de "+cpu_label)
    plt.ylabel("Temps (s)")
    plt.legend(title="Threads (vertical)")
    plt.savefig("fig/times_scal.pdf", transparent=True, bbox_inches='tight')
    plt.show()

plot_times_parallel(lambda p,t : p, "processus")
plot_times_parallel(lambda p,t : p*t, "coeurs")
```

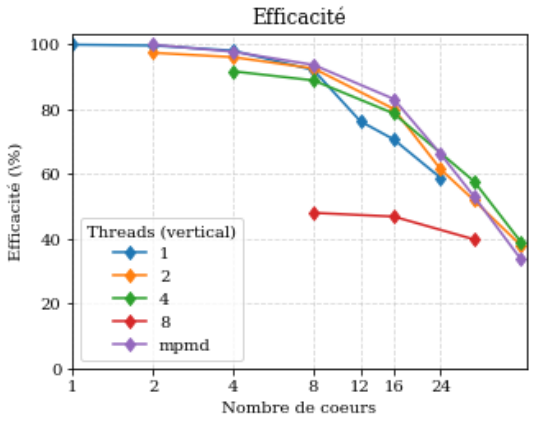




In [9]:
```
def plot_speedup_parallel() :
    plt.figure(figsize=(0.33*textwidth, figheight))
    plt.plot([1,48],[1,48],'--')
    threads = np.array(sorted([a for a in set([t for (p,t) in times_scal.keys()
])]))
    for thread in threads:
        procs = np.array(sorted([p for (p,t) in times_scal.keys() if t==thread]
))
        times = np.array([times_scal[1,1]/times_scal[p,thread] for p in procs])
        plt.plot(procs*thread, times, 'd-', label=str(thread))
    plt.xlim(1)
    #plt.ylim(0)
    plt.xscale('log',basex=2)
    plt.yscale('log',basey=2)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%i'))
    plt.gca().yaxis.set_major_formatter(FormatStrFormatter('%i'))
    plt.title("Speedup MPI+OpenMP")
    plt.xlabel("Nombre de coeurs")
    plt.ylabel("Speedup")
    plt.legend(title="Threads (vertical)")
    plt.savefig("fig/times_scal.pdf", transparent=True, bbox_inches='tight')
    plt.show()

plot_speedup_parallel()
```



In [10]:
```
def plot_eff_parallel() :
    plt.figure(figsize=(0.33*textwidth, figheight))

    threads = np.array(sorted([a for a in set([t for (p,t) in times_scal.keys()
])]))
    for thread in threads:
        procs = np.array(sorted([p for (p,t) in times_scal.keys() if t==thread]
))
        times = np.array([times_scal[1,1]/times_scal[p,thread]/(p*thread)*100 f
or p in procs])
        plt.plot(procs*thread, times, 'd-', label=str(thread))

    procs = np.array(sorted([p for (s,p) in times_mpmd.keys() if s=="double"]))
    times = np.array([times_mpmd["double",p] for p in procs])
    plt.plot(2*procs, times[0]/times*100, '-d', label="mpmd")

    plt.xlim(1)
    plt.ylim(0)
    plt.xscale('log',basex=2)
    plt.xticks(procs)
    plt.gca().xaxis.set_major_formatter(FormatStrFormatter('%i'))
    plt.gca().yaxis.set_major_formatter(FormatStrFormatter('%i'))
    plt.title("Efficacité")
    plt.xlabel("Nombre de coeurs")
    plt.ylabel("Efficacité (\%)")
    plt.legend(title="Threads (vertical)")
    plt.savefig("fig/eff_scal.pdf", transparent=True, bbox_inches='tight')
    plt.show()

plot_eff_parallel()
```



Observations

- On observe une efficacité en MPI+OpenMP comparable à celle observable en MPMD, ce qui indique que la performance sur un noeud est limité par la bande passante mémoire
- 8 threads sur la verticale -> performance problématique (Piste : les 8 threads sont regroupés sur des coeurs du meme socket (= même banc mémoire?))