Exercise 1: Optimize for SIMD "double hummer"

Location: /work/DEISA-Training/EXERCISE_1

!! check always *.lst Listing for messages of compiler about SIMDization

A) Files:
daxpytest.f: parallel daxpy (vector-vector) test with time measurement
dgemmtest.f: parallel daxpy (matrix-matrix) test with time measurement

B) Compile&Run: (no SIMD, start)

Modify Makefile :compile with ARCH=-qarch=450 -O3 -qstrict

ignore misalign warnings

> make
> llrun -np 32 -mode VN -exe daxpytest.rts
> llrun -np 32 -mode VN -exe dgemmtest.rts

--> run time and Mflops rate

C) Compile&Run: (no SIMD, hot)

Modify Makefile :compile with ARCH=-qarch=450 -O3 -qhot
ignore misalign warnings

D) Compile&Run: (SIMD, hot)

Modify Makefile :compile with ARCH=-qarch=450d -O3 -qhot
modify also daxpytest.f --> reduce number of loops by factor 100
ignore misalign warnings

run daxpytest
--> run time and Mflops rate

modify daxpytest.f --> correct missalignment
--> increase number of loops by factor 100
modify dgemmtest.f --> correct missalignment

run dgemmtest
--> run time and Mflops rate

E) Compile&Run: (SIMD, hot, modsrc, O5)

Modify Makefile :compile with ARCH=-qarch=450d -O5 -qhot

--> run time and Mflops rate

F) Compile&Run: (SIMD, strict, MatMul, O3)

Modify dgemmtest.f: switch to other calculation of c(i,j) in line 37

Modify Makefile :compile with ARCH=-qarch=450d -O3 -qstrict
run dgemmtest.rts

--> run time and Mflops rate

G) Compile&Run: (SIMD, hot, MatMul, O3)

Modify Makefile :compile with ARCH=-qarch=450d -O3 -qhot

run dgemmtest.rts

--> run time and Mflops rate

H) Compile&Run: (SIMD, hot, MatMul, O5)

Modify Makefile :compile with ARCH=-qarch=450d -O5 -qhot

run dgemmtest.rts

--> run time and Mflops rate

H) Compile&Run: (SIMD, hot, MatMul, O5, ESSL)

Modify Makefile :compile with ARCH=-qarch=450d -O5 -qhot -qessl -lesslbg

run dgemmtest.rts

--> run time and Mflops rate

Exercise 2: Using tools

Location: /work/DEISA-Training/EXERCISE_2

A) C example: sor.c

compile with
- mpicc -O3 sor.c -o sor -lm # uses gcc
- mpixlc -O3 -qstrict sor.c -o sor -lm # uses xlc

copy matching input file for # numbers of processor (sor.in.#) to sor.in

llrun -np # -mode VN sor

B) Fortran example: swim.f

compile with
- mpif90 -O3 swim.f -o swim # uses gfortran
- mpixlf77 -O3 -qstrict swim.f -o swim # uses xlf

llrun -np # -mode VN swim

Exercise 3: Running MPI Benchmark IMB

Location: /work/DEISA-Training/EXERCISE_3

A) Installation:

Step 1: extract tar file IMB_3.1.tgz in $HOME or a directory below

```
> tar xzvf IMB_3.1.tgz

Readme file:
Readme File for Intel(R) MPI Benchmarks 3.1 Distribution

After unpacking, the directory will contain

  - This file (ReadMe_first)

and 4 subdirectories

  - ./doc  (ReadMe_IMB.txt; IMB_ug-3.1.pdf, the methodology description)
  - ./src  (program source code and Makefiles)
  - ./license (source license agreement, trademark use license agreement)
  - ./versions_news (version history and news)

>>>>
Please read the license agreements first:
  - license.txt specifies the source code license granted to you
  - use-of-trademark-license.txt specifies the license for using the
    name and trademark "Intel(R) MPI Benchmarks"
<<<<

For a quick start on compiling and using the Intel(R) MPI Benchmarks,
see ./doc/ReadMe_IMB.txt.
```

Step 2: Create the executable

modify src/make_mpich

```
> make -f make_mpich

--> IMB-MPI1
```

C) Running the MPI Benchmark:

run pepc

```
llrun -np 32 -mode VN -exe IMB-MPI1  -args "-mem 200"
```

D) Run the Multi-Version of different MPI operation on different numbers of processors using different mode (VN,SMP,DUAL) and compare the timings and bandwidths.

see documentation of IBM in ./doc

Exercise 4: Running application PEPC

Location: /work/DEISA-Training/EXERCISE_4

A) Installation:

Step 1: extract tar file pepc_trunk.tar in $HOME or a directory below

```
> tar xvf pepc_trunk.tar
```

Using ls you will see different files and folders in the new directory. These are the most important ones:

1. lpepcsrc - The heart of PEPC. It contains the tree code and diagnose tools.
2. pepc-b - Plasma Physics. This is the first frontend.
3. pepc-e - Electrostatics. The second frontend demonstrates the use of PEPC as a blackbox.
4. demo_mb - Let it run. This folder contains different job scripts for JUMP and JUGENE.

Step 2: To create the build-in makefiles run

```
> configure
```

which should set up environment variables and makefiles according to JUGENE.

Here are some useful options:

* OPT="-Ox" with x = 0,...,5 switches optimization level (default:-O3)
* OPT="-g -qfullpath" sets debug mode (useful for TotalView)
* --with-visit turns on visualization mode
* --help gives a details overview of possible options and switches

If you want to influence the behavior of this configuration (e.g. create a new directory with source code), you have to edit the file configure.in accordingly and run

```
> autoconf
```

At the end of this step you should have makefiles in the main directory and in the subdirectories lpepcsrc, pepc-b and pepc-e.

Step 3:

Run

```
> make
```

which generates different version of pepc in the directory ./bin.
We will use pepc-b.

B) Input preparation:

It will prove useful to include the PEPC directory in your path, i.e.:

```
> export PATH=$PATH:$PEPC/bin
```

Change to directory $PEPC/Prace_Tutorial

generated directories needed by PEPC:

> create_pes.pl . 512

--> generates sub-directories for 512 MPI tasks

PEPC Input file:
cp laser+slab.h run.h

C) Running the code:
run pepc

llrun -np 512 -mode VN -exe ../bin/pepcb_bgp

D) Test with different parameter sets (see documentation of PEPC)
and different MPI tasks number
--> measure time

E) Instrument PEPC with HPM
see Exercise 2

F) Run PEPC with Scalasca

G) 2D-Visualisation:
gmt based, TBD ...