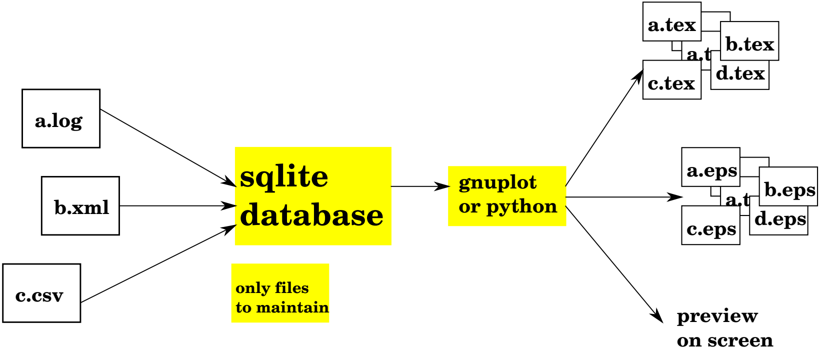
How to make TeX files with tons of graphs (part II)

Context

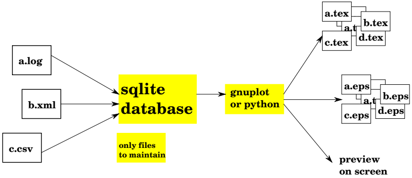
- You want to create a TeX file with lots of graphs easily
- You want to *automate the entire process*, from running experiments to producing TeX document with graphs, so you can *painlessly repeat the experiment* and update data in the document
- With ad-hoc solutions, your directory easily screws up with too many data files and scripts you never understand a week later

The practice

I previously proposed:



The practice



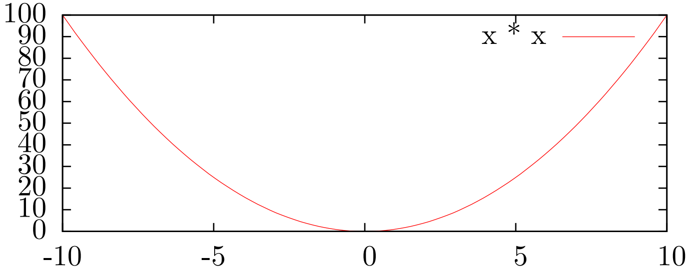
- You maintain only 2 files for all graphs
  - an sqlite3 database that has *all raw data*
  - a gnuplot or a script file that generates data to gnuplot
- I talked about a command, txt2sql, which make it straightforward to convert text files (log) to sqlite3 database
- *Generating lots of graphs from a database was still painful*, and I now address it

What I made this time?

- A small python library to interface with gnuplot command
- There is python-gnuplot package, but I do not rely on it
- A tentative name: `smart_gnuplotter.py`

A simplest example

```
import smart_gnuplotter
g = smart_gnuplotter.smart_gnuplotter()
g.graphs("x*x")
```



Pausing behavior

- It generates “pause -1” so gnuplot waits until you enter a newline
- smart_gnuplotter then asks what to do for the following graphs:

```
’s’ to suppress future prompts, ’q’ to quit, else to continue [s/q/
        other]?
```

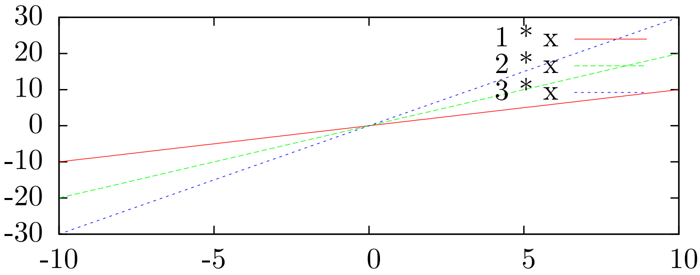
- You may change it by:

```
g.default_pause = 0
```

Writing multiple curves in a single graph

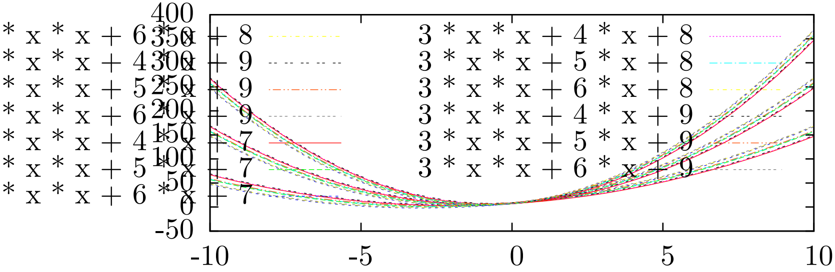
```
g.graphs("%(a)s * x", a=[1,2,3])
```

- The basic form is to parametarize the expression with *%(var)s* and supply its values with *var=list*



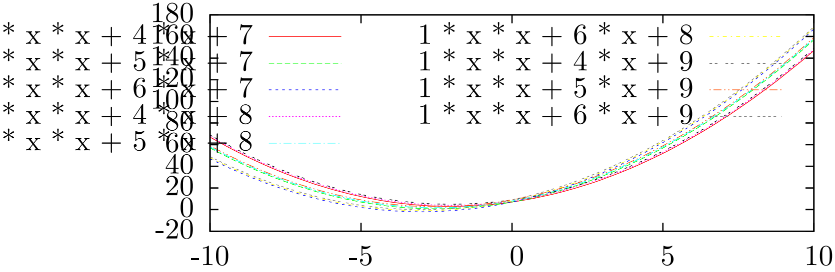
More curves in a single graph . . .

```
g.graphs("%(a)s*x*x+%(b)s*x+%(c)", a=[1,2,3], b=[4,5,6], c=[7,8,9])
```

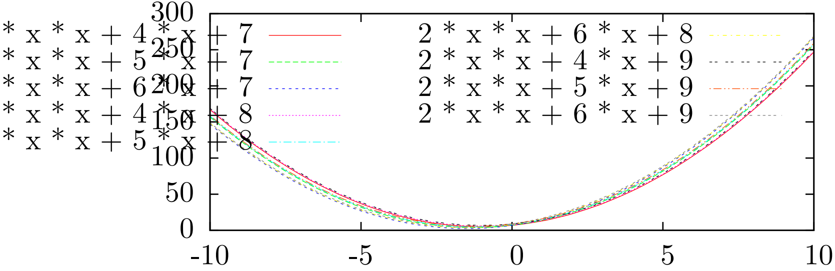


Writing multiple graphs in a single shot (1)

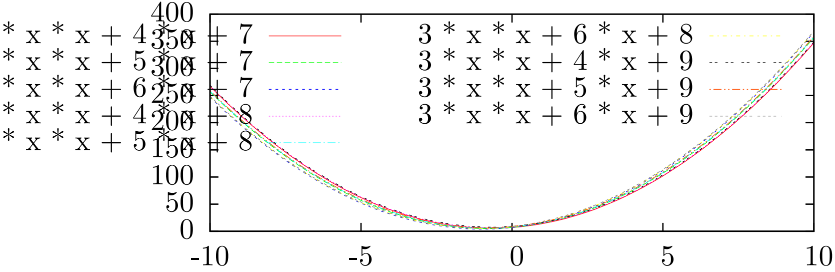
```
g.graphs("%(a)s*x*x+%(b)s*x+%(c)", a=[1,2,3], b=[4,5,6], c=[7,8,9],
         graph_vars=["a"])
```



Writing multiple graphs in a single shot (2)



Writing multiple graphs in a single shot (3)



Parameters that change together

- So the basic form is:

```
g.graphs("expression", var=values, var=values, ...)
```

- But you might not want to generate all combinations
- For example, you may want to have $(a, b) = (\text{"}hoge\text{"}, 10), (\text{"}bar\text{"}, 20)$, not $(\text{"}hoge\text{"}, 20)$ or $(\text{"}bar\text{"}, 10)$
- You may do so by having a parameter assuming tuple values:

```
X=[("hoge",10),("bar",20)]
```

and refer to them as %(X[0])s, %(X[1])s, etc.

Looking what's going on

```
g.graphs(..., gpl_file="hoge.gpl")
```

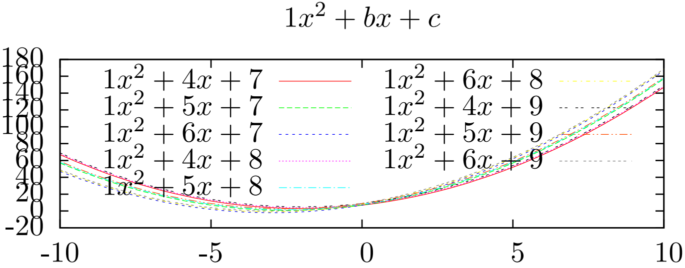
and look at hoge.gpl:

```
set terminal wxt
plot x**x
pause -1
```

- Specify gpl_file and you will get the file given to gnuplot
- When something went wrong, looking at this file is often the simplest way to diagnose

Specifying various attributes

```
g.graphs("%(a)s*x**x+%(b)s*x+%(c)s", a=[1,2,3], b=[4,5,6], c=[7,8,9],
          graph_vars=["a"], graph_attr=r'''
set title "$%(a)sx^2+bx+c$"
''',
          curve_attr=r'''title "$%(a)sx^2+%(b)sx+%(c)s$"''')
```



Specifying various attributes

- graph_attr is whatever comes before the 'plot' command
- curve_attr is whatever comes after each curve

```
g.graphs(E,
         graph_attr=graph_attr,
         curve_attr=curve_attr)
```

```
graph_attr
plot E curve_attr, E curve_attr, ...
```

Shortcut for frequently used attributes

Frequently used attributes can be directly set by keyword arguments

```
g.graphs(..., var=..., var=..., ...)
```

- Graph attributes
  - graph_title
  - terminal
  - output
  - xrange, yrange
  - xlabel, ylabel
  - boxwidth
- Curve attributes
  - curve_title
  - curve_with

Other things to plot

- As it is normal in gnuplot, you may plot
  - datafile

```
g.graphs('"filename"', ...)
```

  - output of a command

```
g.graphs('">cmd"', ...)
```

- Besides, you may plot
  - data in python list

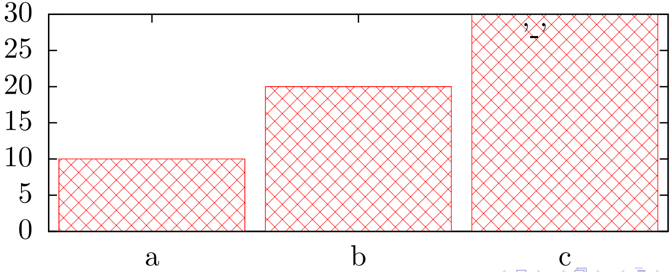
```
g.graphs([(1,2),(3,4),...], ...)
```

  - output of an SQLite query (later)

Bonus : writing graphs with symbolic x-axis

- It's tedious to show a graph with symbolic x-axis
- smart_gnuplotter does all the work you need to do for it

```
g.graphs([("a",10), ("b",20), ("c", 30)],
         yrange="[0:]",
         boxwidth="0.9 relative",
         curve_with="boxes fs pattern 1")
```



Interecting with sqlite3: where the real power comes from

```
g.graphs((database, query, init_string, init_file, udfs, udas,
        udcs), ...)
```

- When you give a tuple as the first argument, a database query is executed and the result treated as data
- init_string, init_file, udfs, udas, udcs are optional
- The above ≈

```
co = sqlite3.connect(database)
for name,arity,f in udfs:
  co.create_functions(name,arity,f)
for name,arity,f in udas:
  co.create_aggregates(name,arity,f)
for name,f in udcs:
  co.create_collations(name,f)
co.execscript(init_string)
co.execscript(content of init_file)
co.execute(query)
g.graphs(co.fetchall(), ...)
```

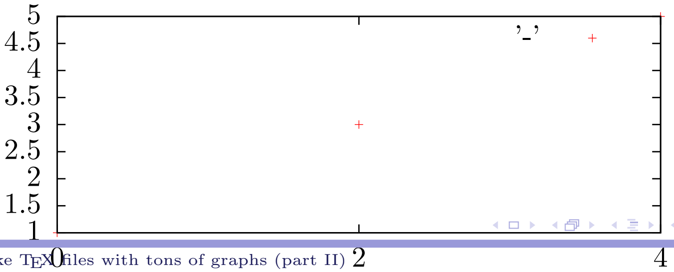
A basic example with sqlite3

Say a database example.sqlite contains the following table

```
select * from t
0|1
2|3
4|5
```

Then

```
g.graphs(("example.sqlite", "select * from t"))
```



A real example (1)

Database 'a' contains all results from matrix multiply

```
sqlite> .schema
CREATE TABLE a (arch, type, ppn, M, gflops_per_sec, ...);
```

- arch : architecture (barcelona, nehalem, sandybridge)
- type : program type (serial, MassiveThreads, Cilk)
- ppn : number of cores (1, 2, 4, 6, . . .)
- M : matrix size
- gflops_per_sec : performance (GFLOPS)

Experiments are repeated, so there are many date of the same (arch, type, ppn, M)

A real example (2)

Let's say we would like to show three graphs:

- serial: compares serial performance among program types, for each architecture
- gflops: shows GFLOPS with cores, for each architecture and program type
- speedup: shows speedup with cores, for each architecture and program type

Serial

- STEP 1: take some time interacting with sqlite3 to come up with a right query showing data for a single graph

```
select type,avg(gflops_per_sec) from a
where ppn=1 and arch="nehalem" and M=704
group by type
```

- STEP 2: parameterize it:

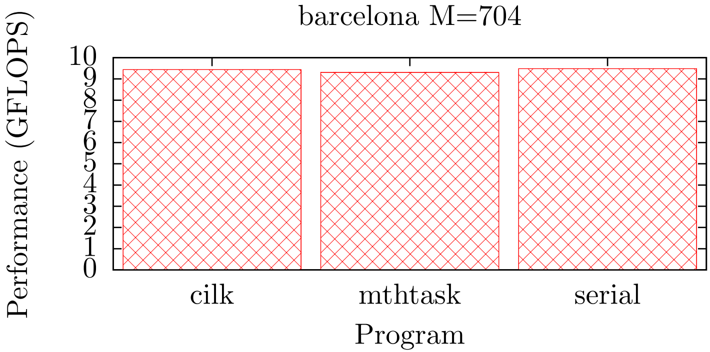
```
g.graphs(("matrix.sqlite",
          r'''select type,avg(gflops_per_sec) from a
where ppn=1 and arch="%(arch)s" and M=%(M)s
group by type'''),
         arch=["barcelona", "nehalem", "sandybridge"],
         M=[704, 1088])
```

do_sql method for a smarter parameterization

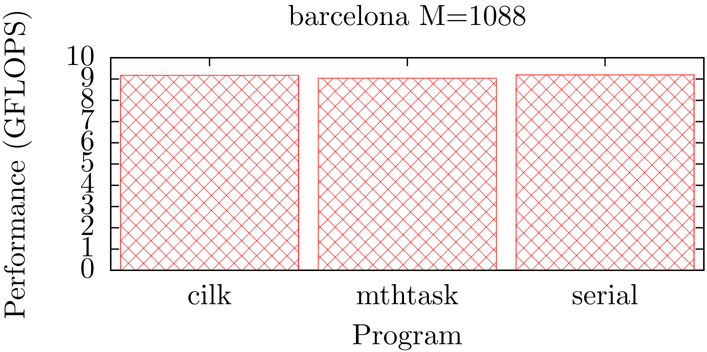
- You often want to ask database to determine parameters
- do_sql method just does that

```
Ms = g.do_sql(db, "select distinct M from a where M > 500", single_col=1)
archs = archs = g.do_sql(db, "select distinct arch from a", single_col=1)
g.graphs(("matrix.sqlite",
          r'''select type,avg(gflops_per_sec) from a
where ppn=1 and arch="%(arch)s" and M=%(M)s
group by type'''),
         arch=archs, M=Ms)
```

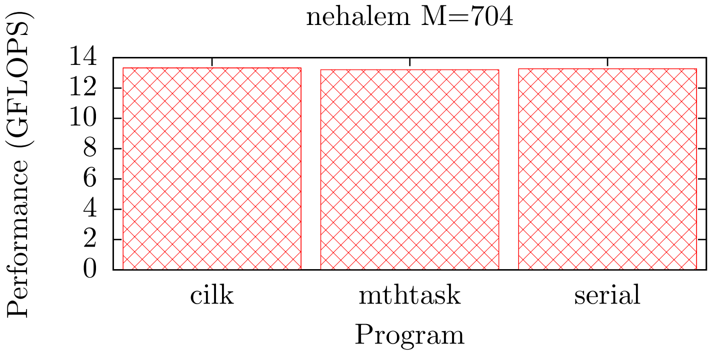
Barcelona, $M = 704$



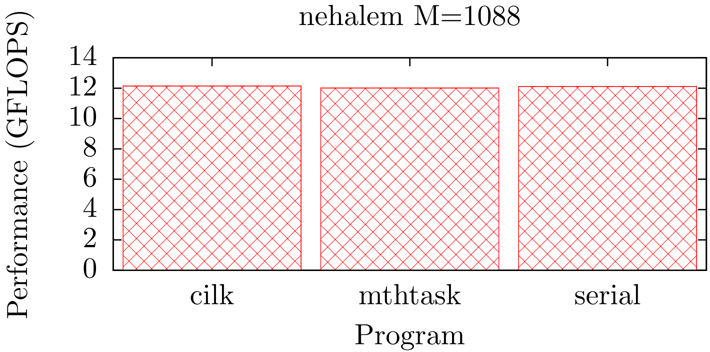
Barcelona, $M = 1088$



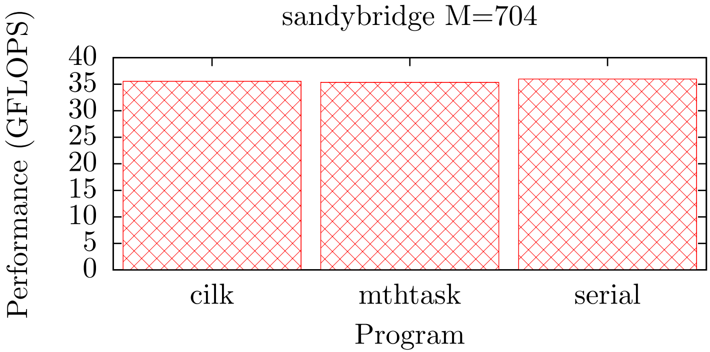
Nehalem, $M = 704$



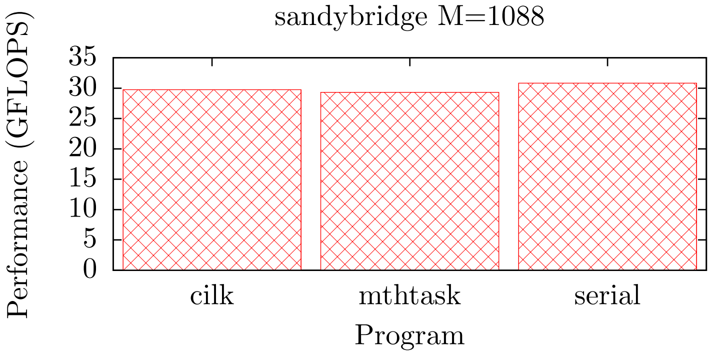
Barcelona, $M = 1088$



Sandy Bridge, $M = 704$



Sandy Bridge, $M = 1088$



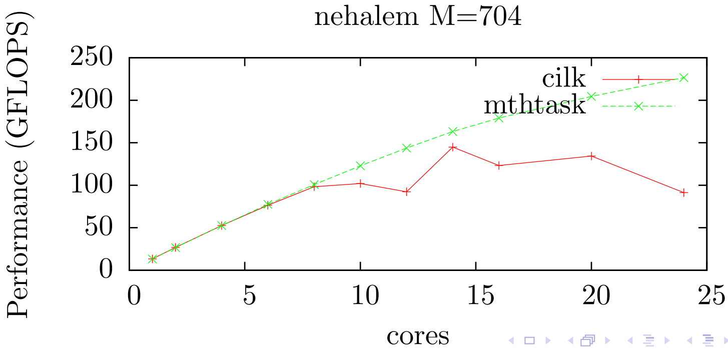
The real code that saves the 6 graphs

```
query = r'''select type,avg(gflops_per_sec) from a
where M = %(M)s and ppn = 1 and arch = "%(arch)s"
group by type'''

g.graphs((db, query),
         output="graphs/serial_%(arch)s_%(M)s.tex",
         curve_title="",
         curve_with="boxes fs pattern 1",
         boxwidth="0.9 relative",
         graph_title="%(arch)s M=%(M)s",
         yrange="[0:]",
         xlabel="Program",
         ylabel="Performance (GFLOPS)",
         M=Ms, arch=archs, graph_vars=[ "arch", "M" ])
```

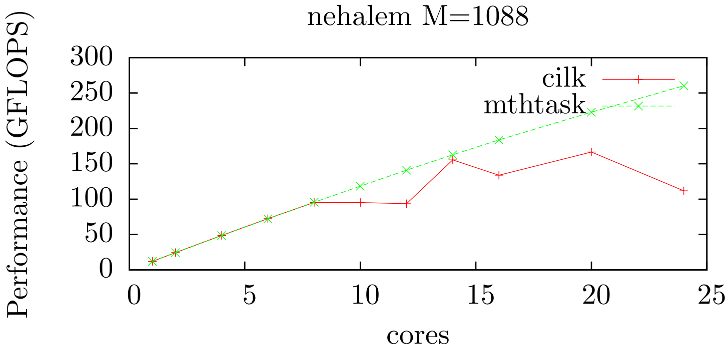
GFLOPS with cores

```
select ppn,avg(gflops_per_sec) from a
where type="%(typ)s" and M=%(M)s and arch="%(arch)s"
group by ppn
```

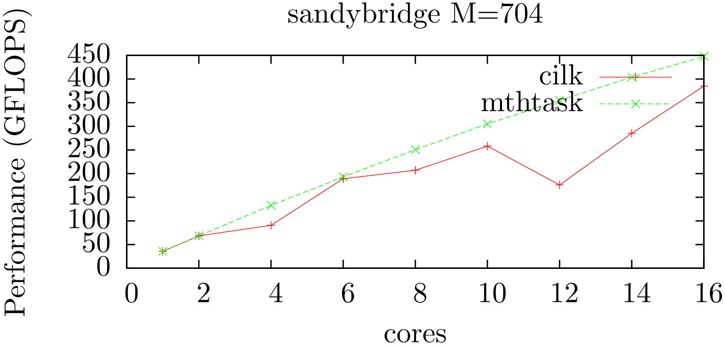
Note: I didn't bind workers to cores for Cilk



Nehalem, $M = 1088$

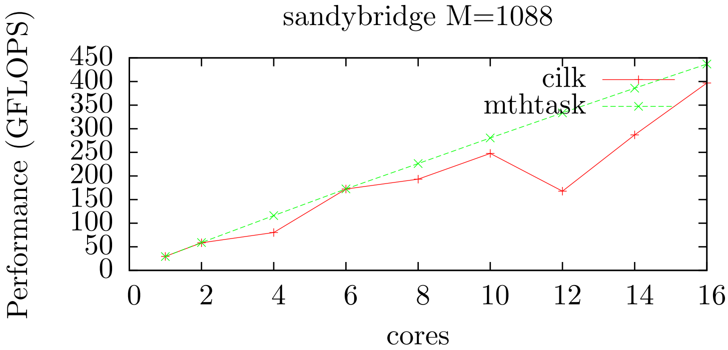


Sandy Bridge, $M = 704$



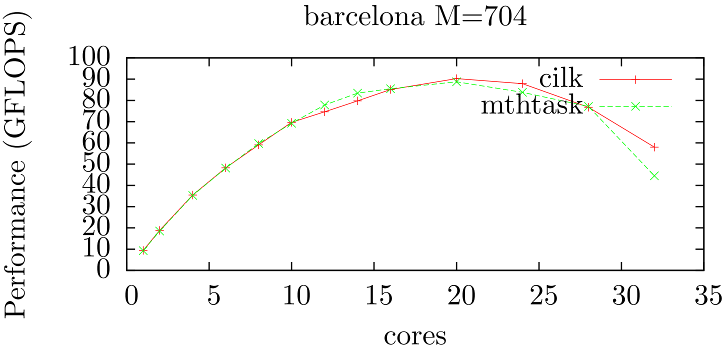
Sandy Bridge, $M = 1088$

Note: TurboBoost is probably on

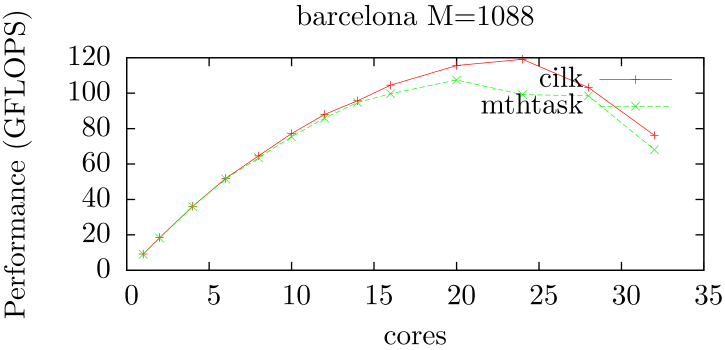


Barcelona, $M = 704$

Note: TurboBoost is probably on



Barcelona, $M = 1088$



The real code that shows the 6 graphs and save them

```
Ms    = g.do_sql(db, "select distinct M from a where M > 500",
         single_col=1)
archs = g.do_sql(db, "select distinct arch from a", single_col=1)
para_types = g.do_sql(db, "select distinct type from a where ppn > 1",
                      single_col=1)

query = r'''select ppn,avg(gflops_per_sec) from a
where type="%(typ)s" and M=%(M)s and arch="%(arch)s"
group by ppn'''

g.graphs(("matrix.sqlite", query),
         output="graphs/gflops_%(arch)s_%(M)s.tex",
         curve_title="%(typ)s",
         curve_with="linespoints",
         graph_title="%(arch)s M=%(M)s",
         xrange="[0:]",
         yrange="[0:]",
         xlabel="cores",
         ylabel="GFLOPS",
         typ=para_types, M=Ms, arch=archs, graph_vars=[ "arch", "M" ])
```

Translating it into speedup

- Speedup is:

$$\frac{\text{GFLOPS of a program}}{\text{GFLOPS of the serial program for the same parameter}}$$

- So the job is to augment the table with a column of the "GFLOPS of the serial program for the same parameter,"

arch	type	ppn	M	gflops_per_sec	serial_gflops_per_sec
nehalem	serial	1	704	50	50
nehalem	mthtask	1	704	48	50
nehalem	cilk	1	704	49	50
sandybridge	serial	1	704	90	90
sandybridge	mthtask	1	704	88	90
sandybridge	cilk	1	704	89	90
...	...	...	...	...	...

Preprocessing

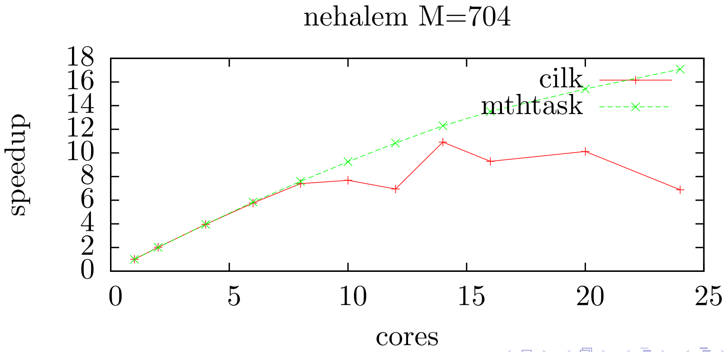
```
create temp table serial as
select arch,M,avg(gflops_per_sec) serial_gflops_per_sec
from a where type = "serial"
group by arch,M;

create temp table b as select * from serial natural join a;
```

Note: you'd better create *temporary* tables only, so you may freely repeat it

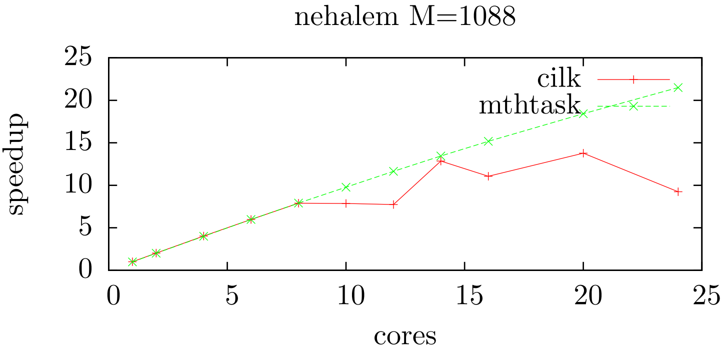
The query

The query itself is easy:

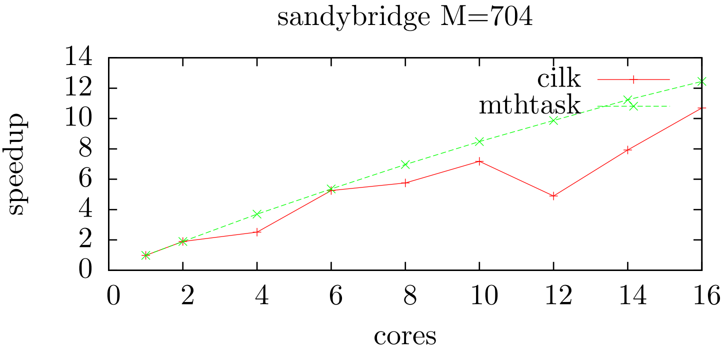
```
select ppn,avg(gflops_per_sec / serial_gflops_per_sec) from b
where type="%(typ)s" and M=%(M)s and arch="%(arch)s" group by ppn
```



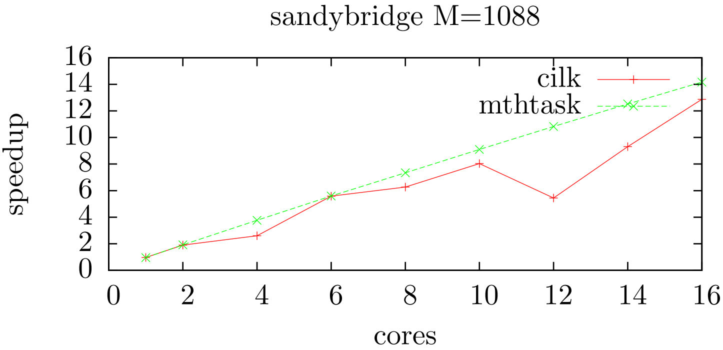
Nehalem, $M = 1088$



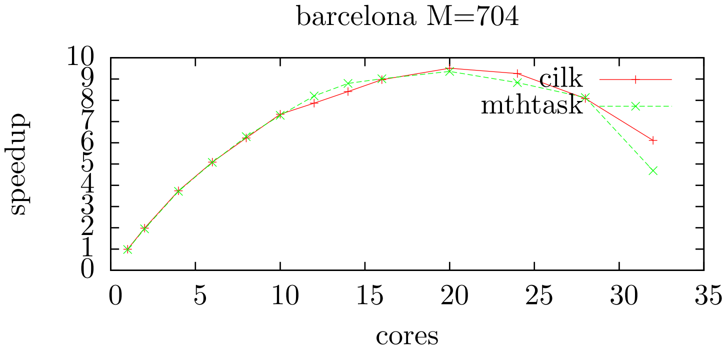
Sandy Bridge, $M = 704$



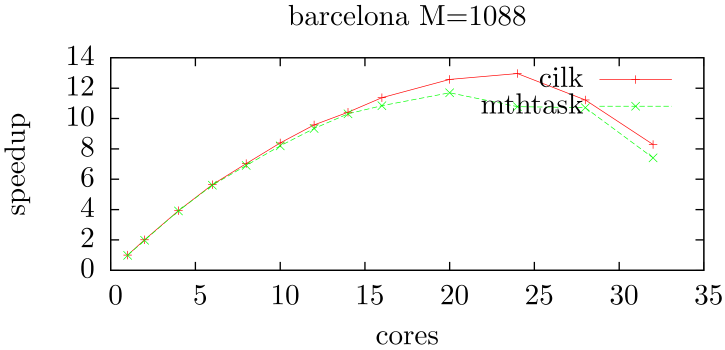
Sandy Bridge, $M = 1088$



Barcelona, $M = 704$



Barcelona, $M = 1088$

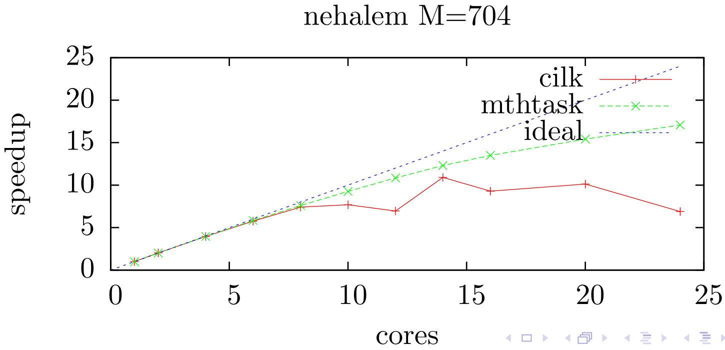


The real code that shows the 6 graphs and save them

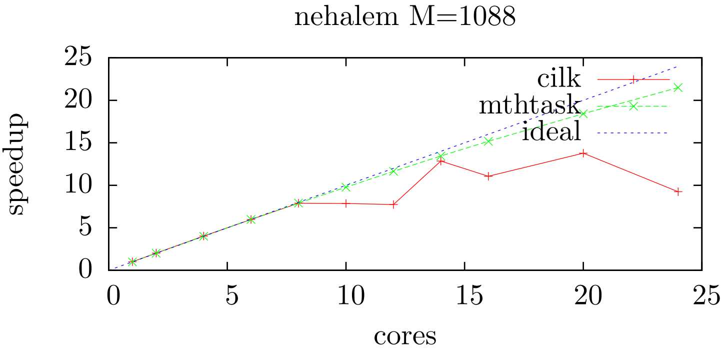
```
init = r'''create temp table serial as
select arch,M,avg(gflops_per_sec) serial_gflops_per_sec
from a where type = "serial"
group by arch,M;

create temp table b as select * from serial natural join a;
'''
query = r'''select ppn,avg(gflops_per_sec / serial_gflops_per_sec) from b
where type="%(typ)s" and M=%(M)s and arch="%(arch)s" group by ppn'''

g.graphs(("matrix.sqlite", query, init),
         output="graphs/speedup_%(arch)s_%(M)s.tex",
         curve_title="%(typ)s",
         curve_with="linespoints",
         graph_title="%(arch)s M=%(M)s",
         xrange="[0:]",
         yrange="[0:]",
         xlabel="cores",
         ylabel="speedup",
         typ=para_types, M=Ms, arch=archs, graph_vars=[ "arch", "M" ])
```

Overlaying “ideal” speedup

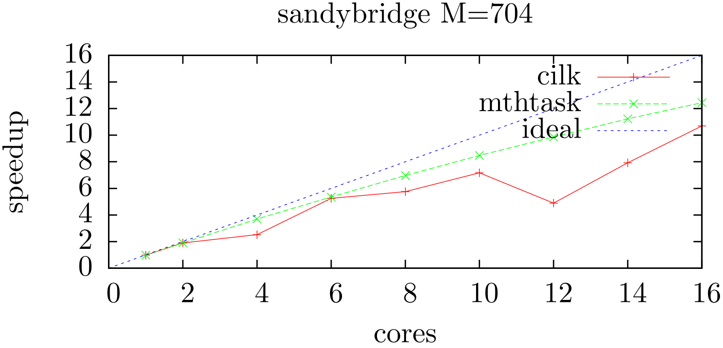
```
g.graphs(("matrix.sqlite", query, init),
         output="graphs/speedup_%(arch)s_%(M)s.tex",
         overlays=[("x", 'title "ideal"')],
         ...)
```



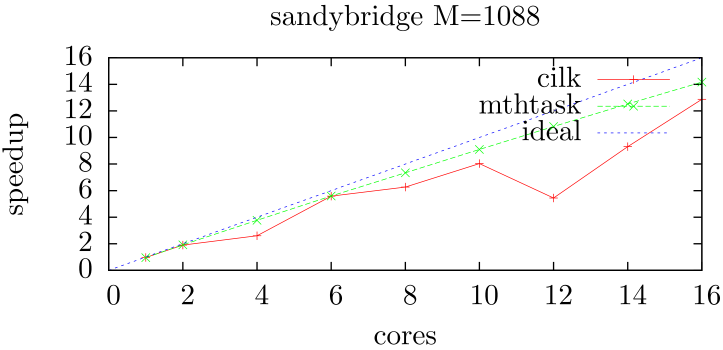
Nehalem, $M = 1088$



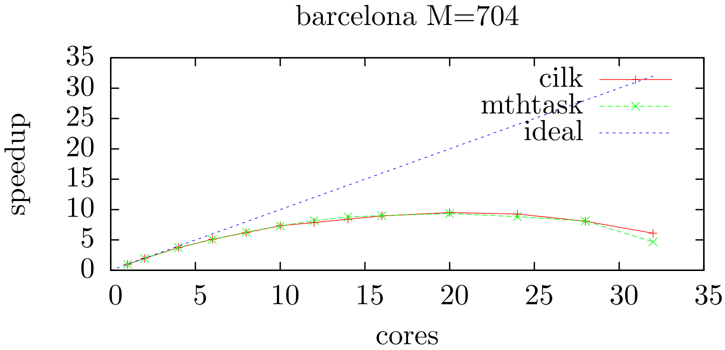
Sandy Bridge, $M = 704$



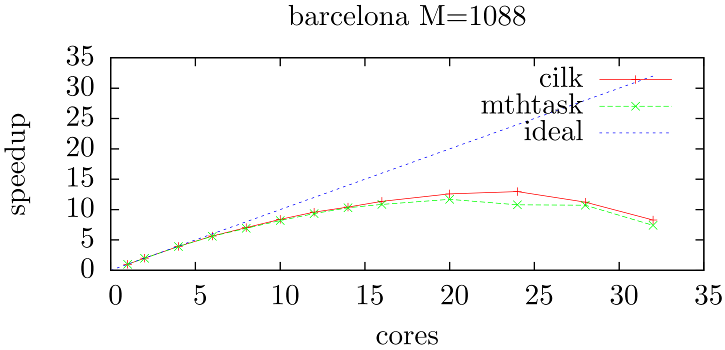
Sandy Bridge, $M = 1088$



Barcelona, $M = 704$



Barcelona, $M = 1088$



Note on the results

- Do not trust Cilk results. It will become significantly better by binding workers to cores
- Now I proved the importance of automating the process!
- Results on Sandy Bridge (hongo600) will have been affected by TurboBoost, which boosts performance on a single core
- We'd better to confirm it by turning TurboBoost off on hongo6xx, but utilizing these dynamic behaviors increasingly sophisticated may be an interesting direction
- Results on Barcelona need true serious investigation and will be a research theme for us

Questions? Objections?