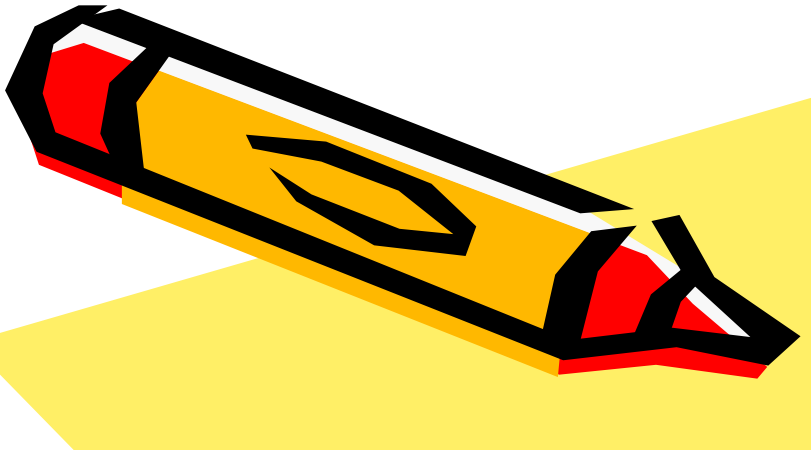


GXP Make

Kenjiro Taura

GXP make

- What is it?
  - Another parallel & distributed make
- What is it useful for?
  - Running jobs with dependencies in parallel
  - Running many long-running jobs where FT is mandatory
  - Managing dynamic jobs generated from changing sources
- Make is a very good “workflow” language and execution engine

Usage

- STEP 1: With GXP, grab (explore) as many hosts as you like

```
$ gxpc use …
$ gxpc explore …
```

- STEP 2: Write a regular Makefile and

```
$ gxpc make -j
```

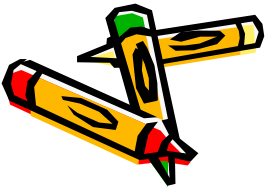
- then tasks are distributed to hosts you explored (**`-j`** option for parallel execution)

Why make is good? (1)

- Concise descriptions
  - A single line can generate
    - tasks with1,000 different parameters
    - tasks with 1,000 different input files
- (almost) automatic parallelization
  - GNU make's `-j` option (but this is for SMP)
- Dependency management
  - Make analyzes what to do from file system states and executes them
  - Parallelization almost without efforts

Why make is good? (2)

- Simple fault tolerance
  - If some jobs fail, run make again
  - Byproduct of dependency management
- Popular and robust implementation
  - Much more popular than any other “future-standard” workflow systems
  - GNU make comfortably manages 10,000 child processes on 1GB memory machine (quality rarely found in research prototypes)



Parallel make is not new

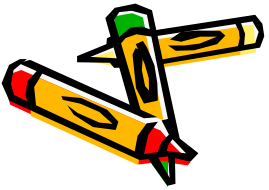
- Sun Grid Engine has qmake
  - Only for SGE
  - Each job takes a job submission through the queue
  - GXP is an “overlay scheduler”
- Other implementations (dmake, etc.)
  - I don’t know any which runs with unmodified GNU make and runs on multiple underlying resource access protocols

Useful Features

- state.html
  - Job/worker status, parallelism profile, etc.
  - Simple heuristics find files in the command lines and they are made anchors

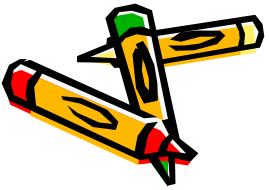
Current Directory

- Rule:
  - If you run gxpc make from directory *D*, then *D* *must exist* on all hosts
    - They do not have to be the same (shared) directory. Just directories of the same path names suffice
  - Commands will run with *D* as CWD
  - You do *not* have to cd to *D* on all hosts



Current Directory

- Special rule:
  - If *D* is under your home directory, it is considered “relative” to your home
  - E.g.,
    - On A: you have /home/tau/abc
    - On B: you have /usr/home/taue/abc
    - If you run gxpc make in A’s /home/tau/abc, then a command would run on B in /usr/home/taue/abc

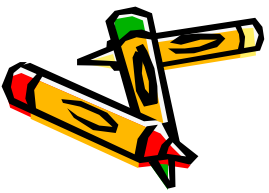


How many tasks per worker?

- (tentative)
  - Write a config file gxpc_make.conf
- Format:
  - **cpu** *regexp n*

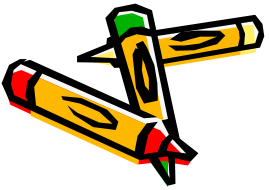
Implementation

- to understand how resources are used and where does the current scalability limit come from



GNU make features essential for GXPC make

- GNU make is a great tool that
  - allows command shell to be replaced (SHELL=…)
  - allows common include files to be specified in environment variables (MAKEFILES)
  - handles >10,000 with no problems



SHELL=

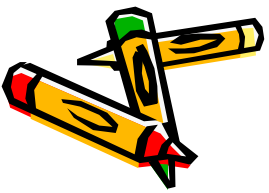
- By writing

```
SHELL=foo
```

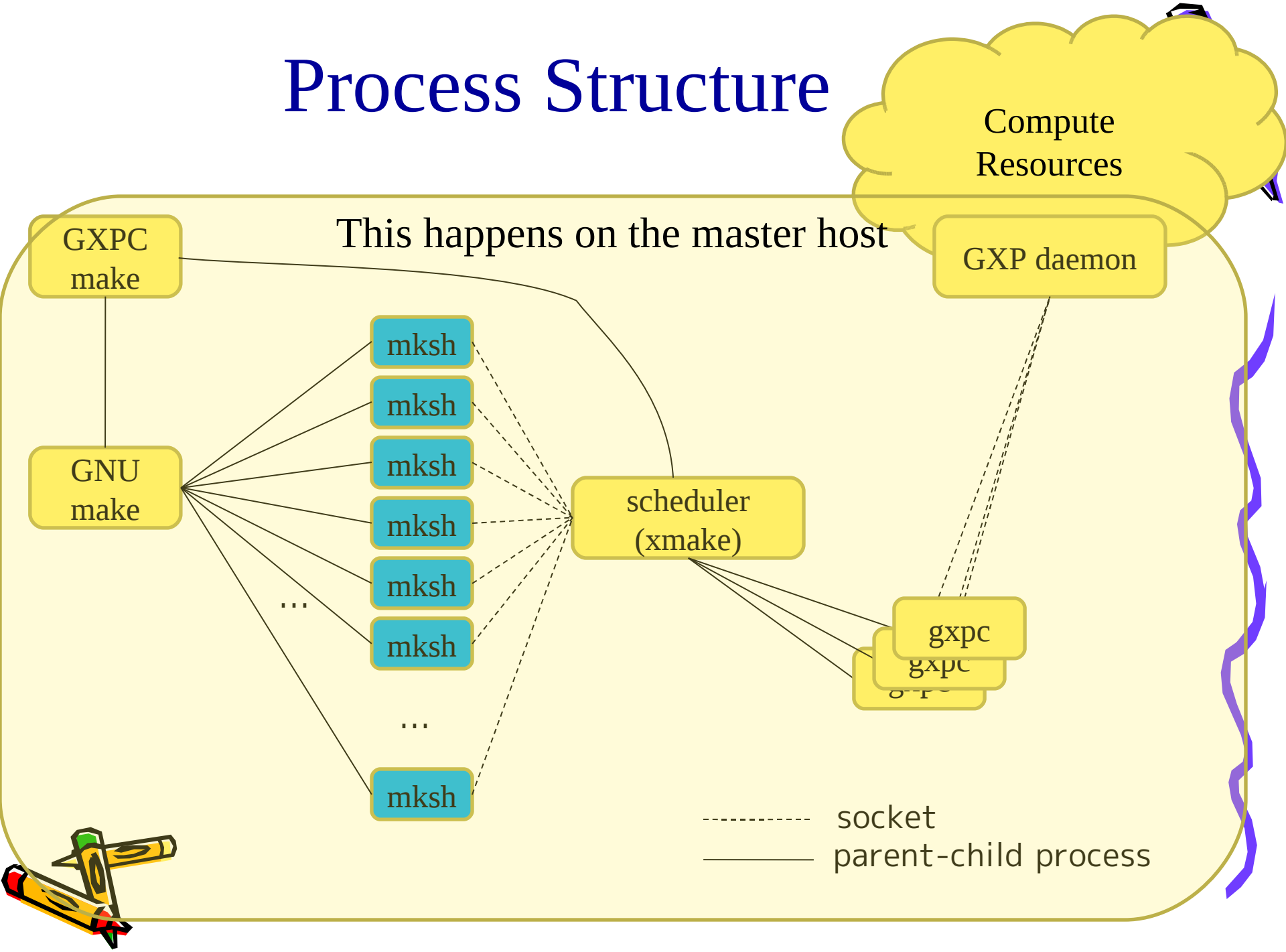
- in your Makefile, make will run **`foo -c`** ***cmd*** to run ***cmd*** (instead of **`/bin/sh -c`** ***cmd***)
- You can effectively “intercept” sub-process invocation of make

MAKEFILES environment

- But the user would have to insert **`SHELL=`** to every single Makefile?
- No, thanks to **`MAKEFILES`** environment variable
- make automatically includes files specified in this variable before every Makefile



Process Structure



How commands run and die

- GNU make spawns a shell (mksh) for command to run
- mksh talks to our scheduler (xmake)
- xmake selects jobs and spawns gxpc to send them to remote
- when a command terminates, xmake sends EOF to the mksh
- mksh exits upon receiving EOF
- GNU make then knows the command finished

Performance/Scalability

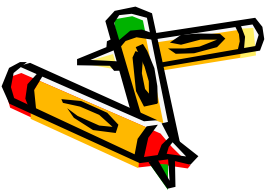
- Master host is a bottleneck
- This seems unavoidable with the approach of using unmodified GNU make
  - Question is *when* we hit the limit
- In our structure,
  - mksh (direct children of make)
  - gxpc
- are the main resource consumers

- Naively,

  `make -j` *`N`*

- with *P* workers will retain
  - *N* mksh processes
  - *P* gxpc processes
- on the master host
  - If mksh and gxpc each takes 1MB, running 1,000 workers eat 2GB

- In our implementation
  - gxpc will terminate without waiting for the remote command to finish (*P* is small)
  - mksh becomes (exec) a tiny process that only waits for a line from stdin and exits
    - if read x; then exit $x; else exit 126;



Performance tips (1)

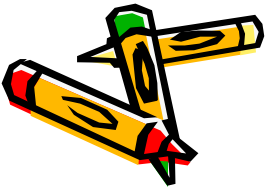
- Use multi-core machines for the master
  - They are good for handling thousands of small processes
- Use `-j` *N* to limit concurrency (to somewhere around worker numbers)

Performance tips (2)

- Load avg on the master gets high (10.0 is norm) but this is not a red signal, per se
- Watch for thrashing (si and so fields)

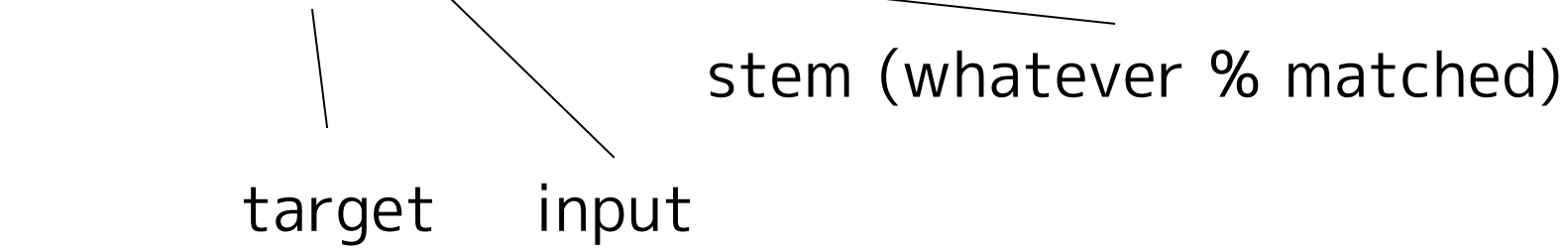
```
vmstat 1
```

- state.html also shows load avg and vmstat, but update is infrequent

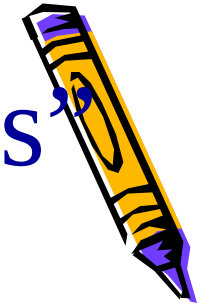


Useful GNU make tips

- Pattern rules (%.xyz : %.pqr)
- $(wildcard …)
- $(shell …)
- $(addsuffix …)
- $@, $<, $*

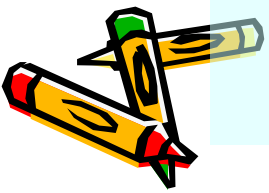


Recursive make for “dynamic jobs”



- With a single invocation of make, it is difficult (impossible?) to have a graph of *dynamic* number of tasks
  - *dynamic* : not known until some commands run
- Example:
  - How to split a large data into fixed size chunks and apply a command to each in parallel?

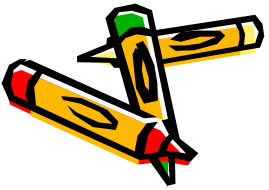
```
Result : big                    This is serial
  split big
  for f in big.*; do … done
```



Recursive make

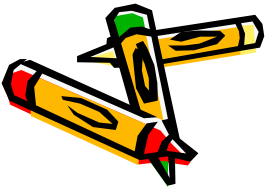
```
result : big
   split big frag.
   $(MAKE) $(subst frag.,int.,
                  $(wildcard frag.*))
   cat int.* > result

int.% : frag.%
   …
```



Issues and extensions for large scale

- Job placement
- Concurrency control
- File sharing
  - Staging
  - File systems



Job placement

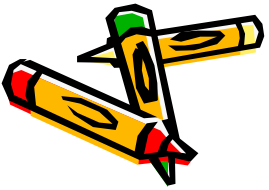
- Currently very limited
- With == before the command, it will run locally. E.g.,

```
main.exe : a.o b.o c.o
    == gcc –o main.exe a.o b.o c.o
```

- will run gcc locally

Concurrency Control

- You may want to limit the concurrency of certain types of jobs. e.g.,
  - < 10 concurrent downloads from a web server
- gxpc make can define “semaphore”s and annotate commands so they acquire some of them
- Notation:
  - =(res=*semaphore_name*) or =(*semaphore name*) before the command



Example

```
%.dat :
    =(fileserv) wget http://foo/big.dat
```

- Then

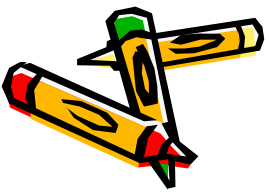
```
gxpc make -- --sem fileserv:5
```

- will run at most 5 wgets in parallel

Default Concurrency

- By default, local jobs (commands with ==) have the concurrency limit of 1
- You can change it with

```
gxpc make -- --sem local:N
```



File sharing

- Within a cluster it is usually not an issue (NFS to some extent or some cluster file system)
- It is a headache across clusters
  - Staging
  - Distributed file systems?
    - Gfarm, hadoop, sshfs-mux

Staging (experimental)

- Only staging from/to the local host is currently supported
- Give `=(in=`*file*`, out=`*file*`, …)` before the command. E.g.,

```
a.dat : a.src
   =(in=$<,out=$@) foo $< > $@
```