



Large Scale Text Processing made simple by GXP make

Kenjiro Taura, Takeshi Shibata,
Yoshikazu Kamoshida, Nan Dun,
Sun Jung Choi, Takuya Matsuzaki,
Jun'ichi Tsujii (University of Tokyo)



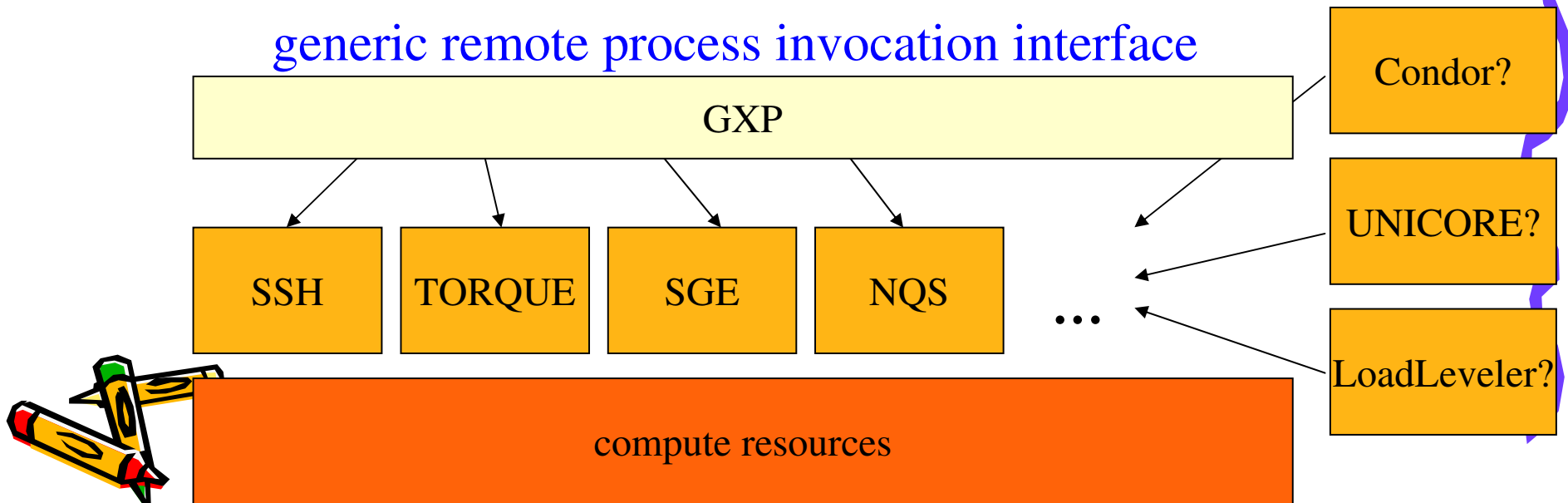
Outline

- GXP and GXP make
 - Demo
- Design goal and philosophy
- Medline to medie workflow experiences
- Scalability limitations and performance



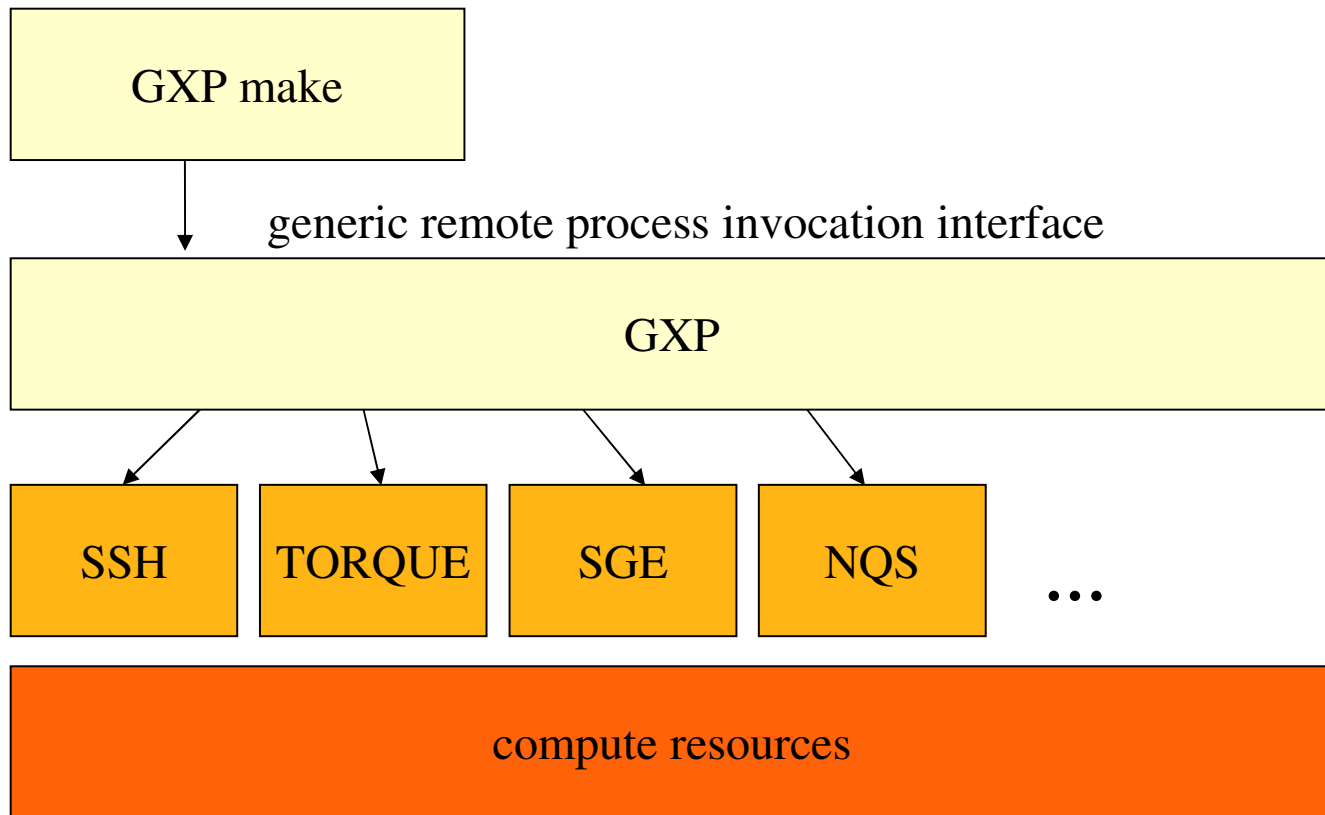
GXP

- A (parallel) process launcher
- Runs on top of various remote exec interfaces. ssh, rsh, batch schedulers (torque, SGE, NQS)
- Exposes a “generic” process invocation interface to upper levels



GXP make

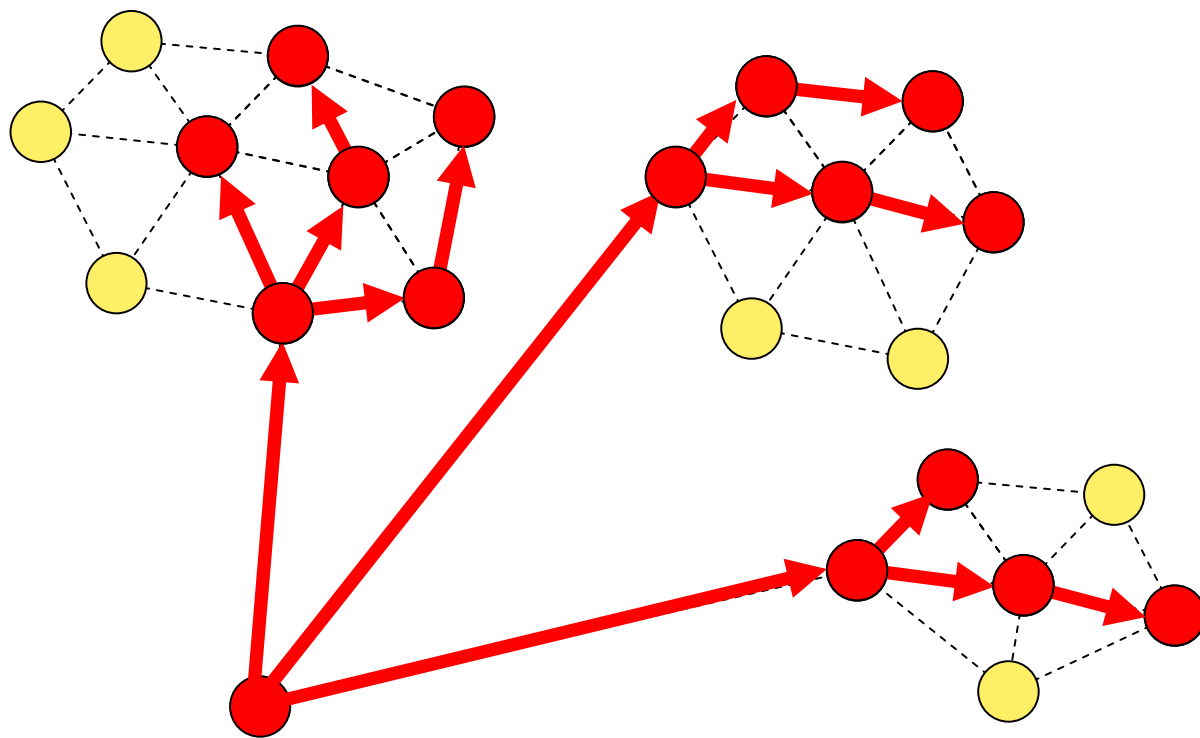
- A parallel and distributed make on top of GXP
- Uses GXP to dispatch jobs to remote resources



Simple GXP demo

1. you say which hosts can login which hosts and how (**use**)
2. say which hosts you want to acquire (**explore**)
3. execute commands (**e** etc.)

You can execute any of them at any time, in any order



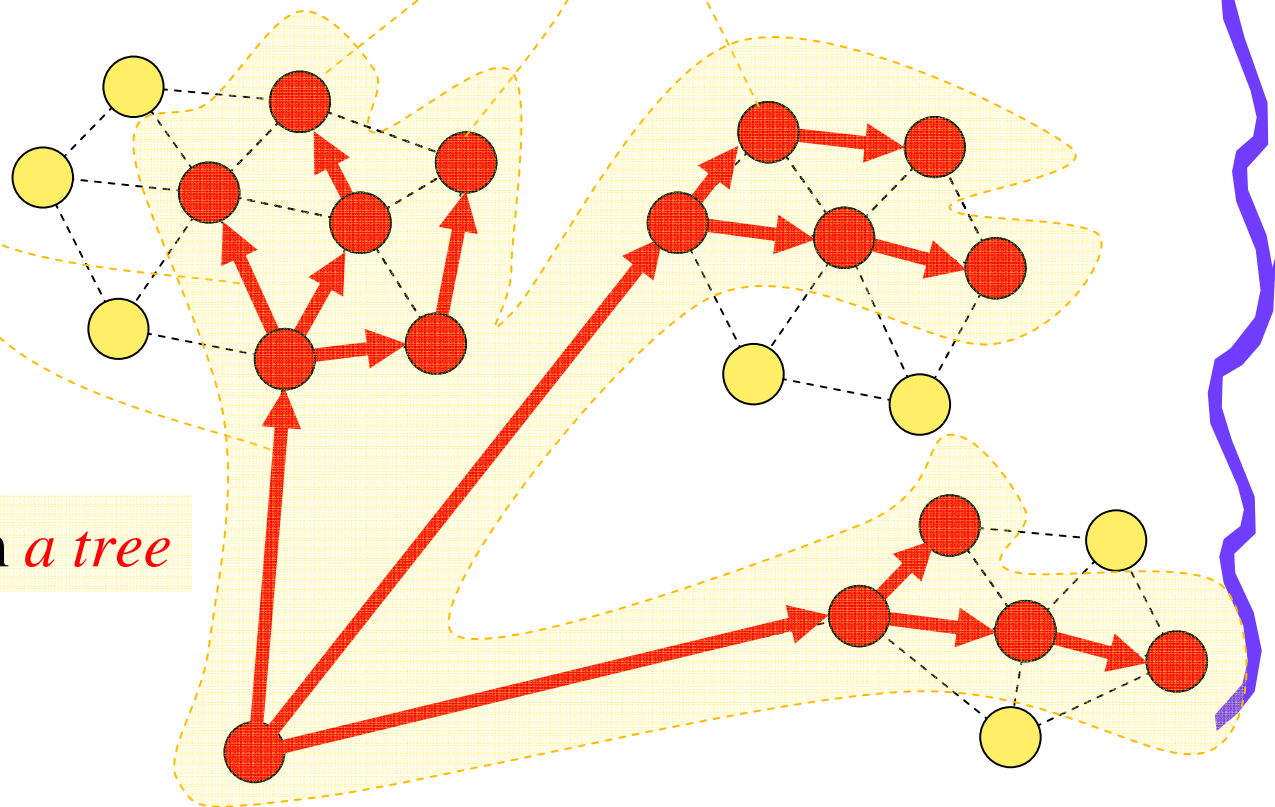
Basic concepts



3. A daemon creates other daemons by an underlying **rsh-like commands** (e.g., SSH, torque, SGE, etc.)

1. GXP *daemons* (*gxpd.py*) stay running

2. Daemons form *a tree*



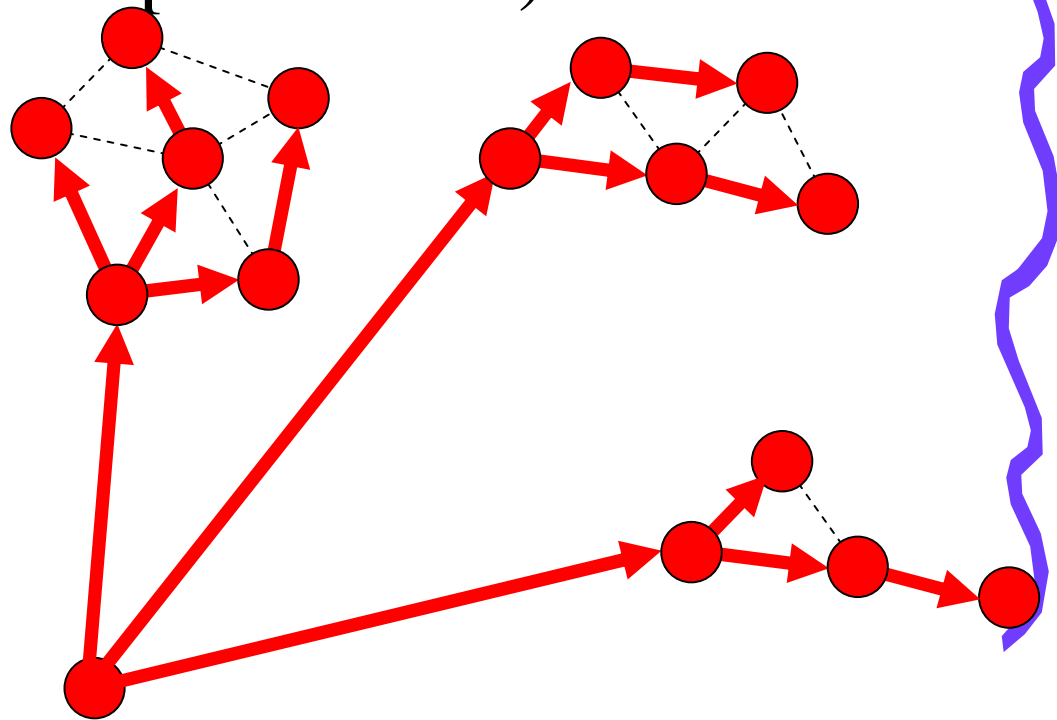
GXP make



- After you grab (i.e. **explore**) resources,

```
$ gxpc make -j N
```

- will read Makefile and dispatch commands to resources (up to N parallelism)



Some important points



- Grabbing resources (**explore**) and running commands (**e**) are separate steps
 - “Job submission” to the underlying resource (via batch scheduler, ssh, etc.) happens only once for a node
 - Subsequent process invocations do not go through it
- Specifying how to grab resources (**use**) and grabbing resources (**explore**) are separate steps
 - Naturally work in heterogeneous environments



Design Goals



- How to minimize the time it takes to produce “**the end result**” of your workflow?
- Enhancing productivity is far more important than squeezing out “**performance**” (machines should work for us, not vice versa)



Where we lose time?



- Diagnosing errors!
 - Your errors in description of tasks
 - Component bugs
 - Job submission errors
 - Authentication errors
 - Middleware bugs (even undeterministic)
 - Down resources

Errors *do* happen. Those are *unavoidable*.

“Reducing chances of human errors and making diagnosis easy must be incorporated in the design”



How GXP (and GXP make) helps enhance productivity?



- GXP explores what would result in “job submission errors” **on the right spot**
 - Once explored, subsequent process invocations are much less likely to fail

- GXP gives you an **interactive** environment over (uninteractive) batch scheduler

- Make gives you a straightforward way to compose arbitrary binaries into workflows, diagnose them interactively (-



They are all Unix principles!



- “shell scripts” and “make” are great “composition” languages

How you would do in programs (i.e. scripts) is no different from how you would do interactively

- No syntactic difference between interactive environment and scripts
- Pipe and redirection : no difference between interactive I/O and file I/O or process communication



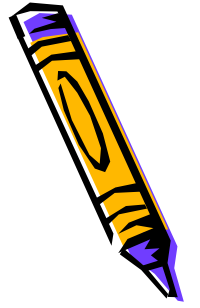
“Parallel/distributed” analogues of Unix principles



- How you do it should be no different,
 - whether **in parallel** or **on a single node** (perhaps interactively)
 - whether **with SSH** or **with batch scheduling systems**
 - whether **across multiple clusters** or **in a single cluster**
- At the same time, anticipate things go wrong with resources so you better not rely on individual resources providing identical interfaces



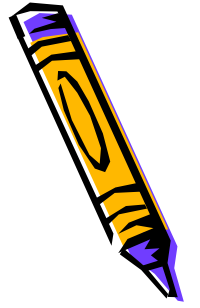
Why make is good ? (1)



- Concise and declarative compositions
- Declarative parallelization
 - You write dependencies, not what to parallelize
 - GNU make's `-j` option (but this is for SMP)
- Dependency management
 - Make analyzes what to do from file system states and executes them



Why make is good? (2)



- Simple fault tolerance
 - If some jobs fail, run make again
 - Byproduct of dependency management
- Popular and robust implementation
 - More proven-to-work than anything else
 - GNU make comfortably manages 10,000 child processes on 1GB memory machine (quality rarely found in research prototypes)



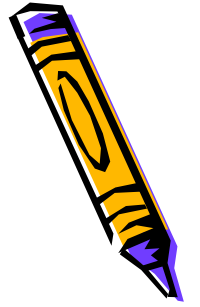
Medline to medie workflow experiences

- Makefile written by Takuya Matsuzaki
- Environments
 - TSUBAME (Titech supercomputer; SGE)
 - InTrigger cluster of clusters (SSH)
 - HA8000 (U-tokyo supercomputer; NQS)
 - Other potential targets are Amazon EC2 and Hector in UK?



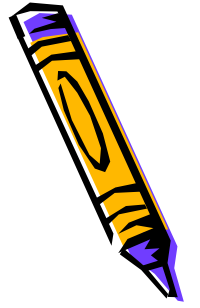
Input and output

- Input : published medline abstracts (767 xml files. 20GB gzipped)
- Output : search index of medie (semantic retrieval engine for medline)



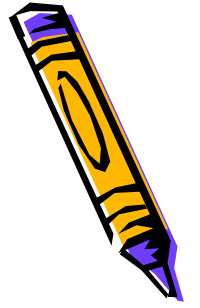
200 files experiences

- Approximately a quarter of the entire abstracts
- Finished in 6 hours with 2,400 CPU cores (Titech TSUBAME)



Implementation of GXP make

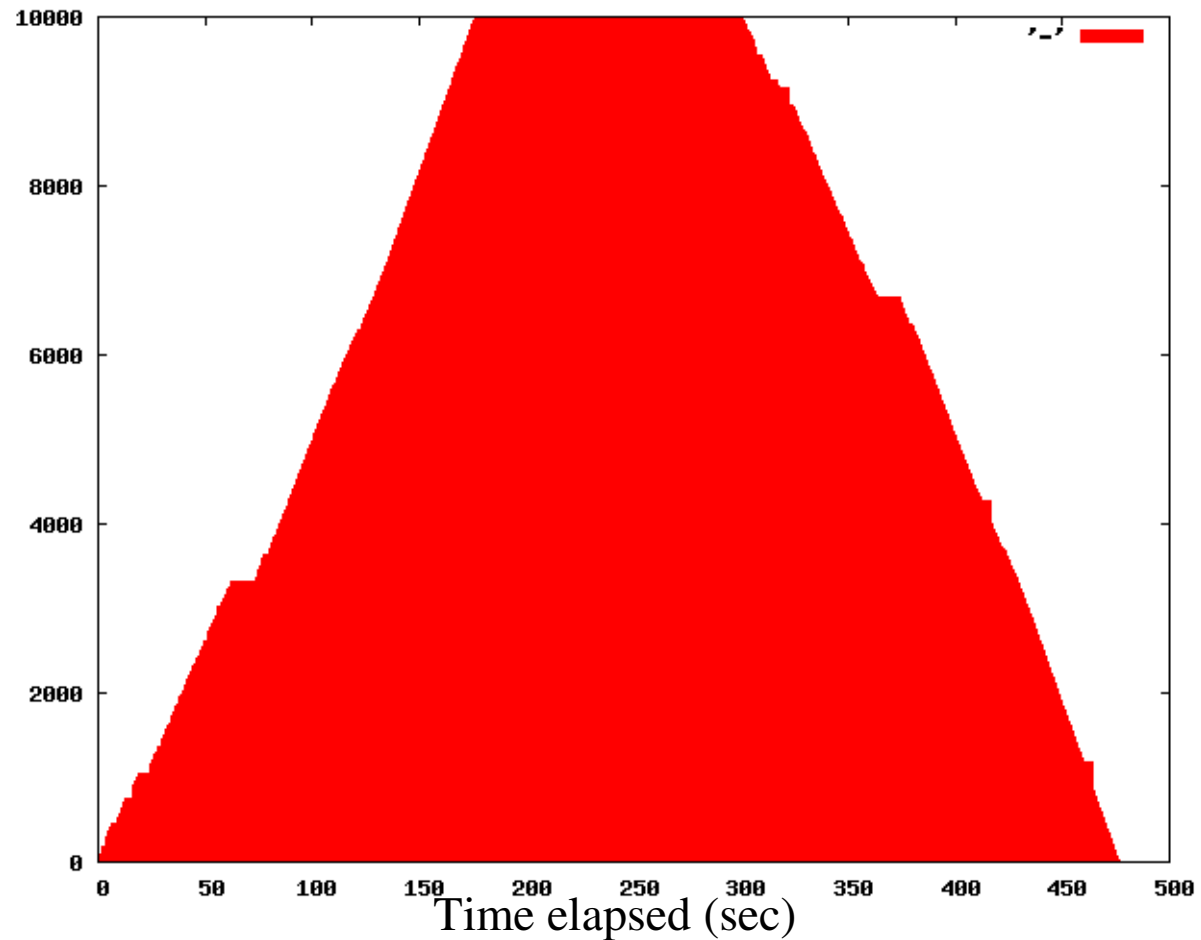
- for the purpose of understanding resource requirements, scalability limits, and achievable speedup



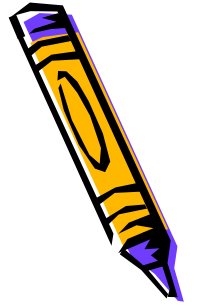
10000 jobs in parallel

- Each job is a simple “sleep 300”
- 148 nodes (up to 80 outstanding jobs per node)

parallelism



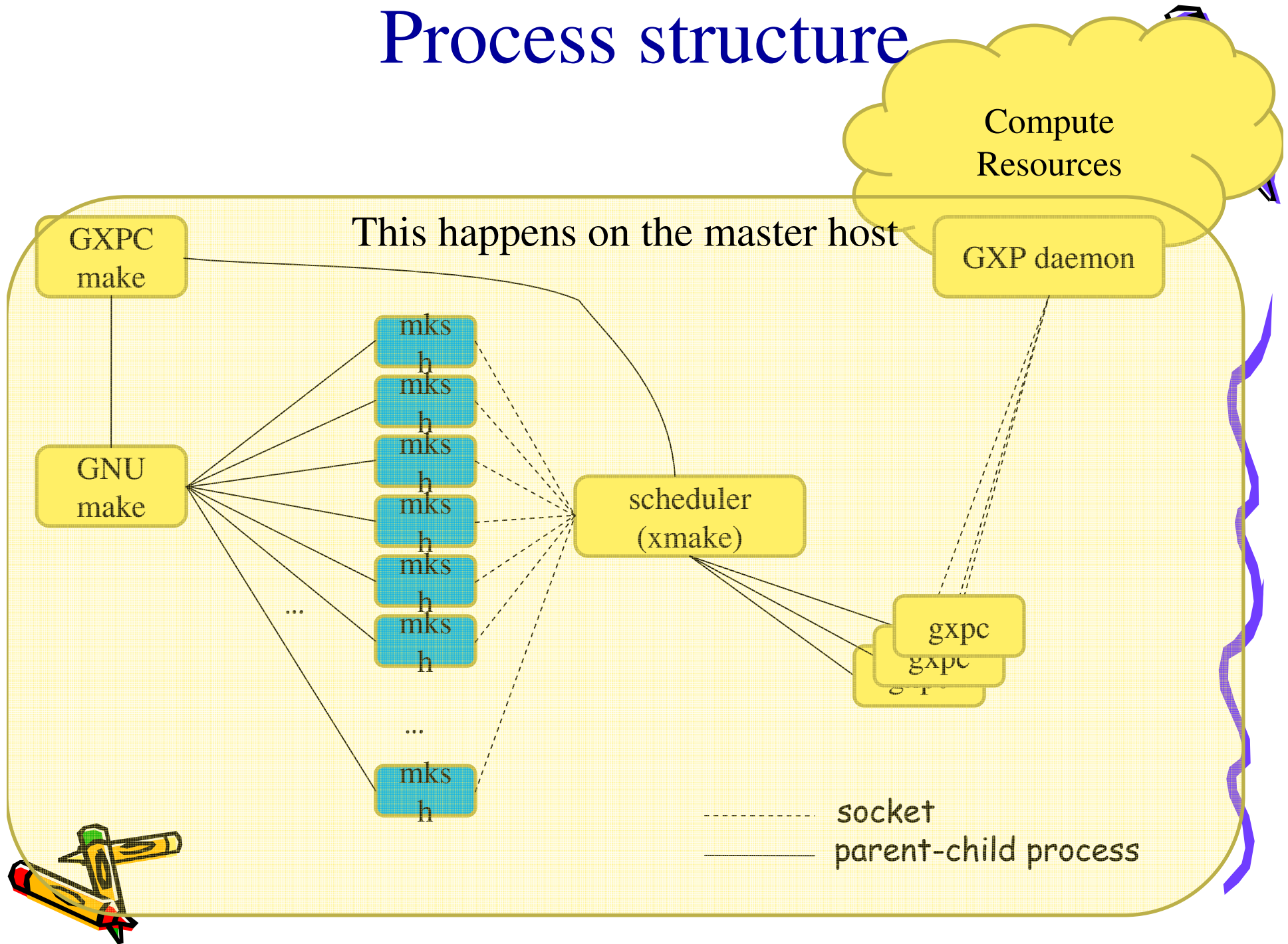
Basic Strategy



- We use GNU make as is (with *no* modification)
- Intercept GNU make's process invocations and queue them to a scheduler (xmake)
- The scheduler then dispatches requests to idle workers



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 and achievable speedup before there
- In our structure,
 - mksh (direct children of make)
 - gxpc



are the main resource consumers



Resources required on the master



- 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 ($P = N = 1,000$) will eat 2GB



Reducing the resource consumption



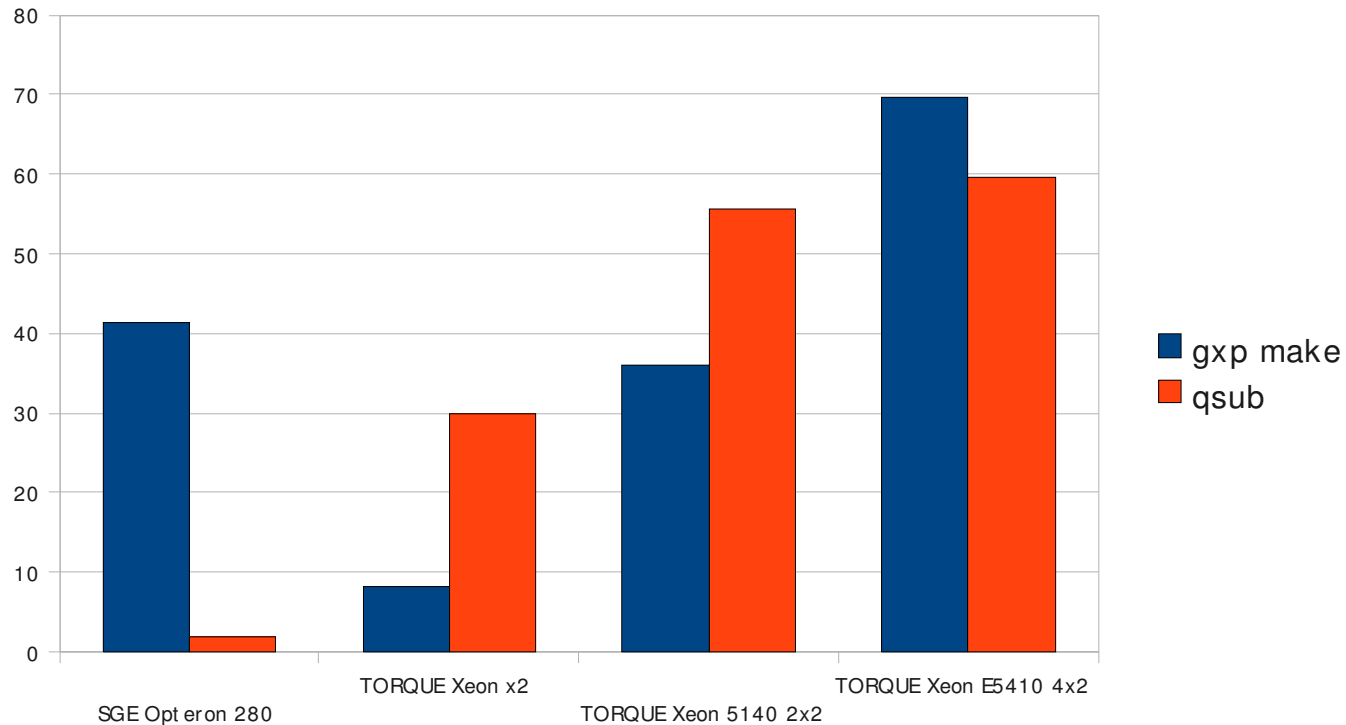
- In our implementation
 - gxpc will terminate without waiting for the remote command to finish (P is small)
 - mksh becomes (exec) a tiny process (🕒 50KB) that only waits for a line from stdin and exits
 - if read x; then exit \$x; else exit 126;



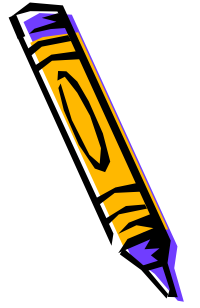
Job submission throughput



- Compared with throughput of the underlying batch scheduler



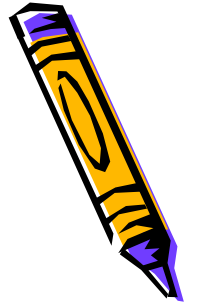
Quantifying speedup (1)



- It is bounded by the rate at which the mater spawns and reaps jobs
 - GNU make: spawns mksh
 - mksh: enqueues job to xmake
 - xmake: sends the job by gxpc
 - xmake: notified of a finished job
 - xmake: disconnect to mksh
 - GNU make: detects mksh's death



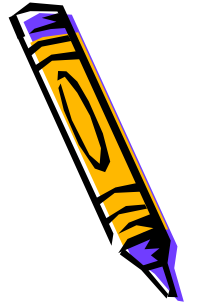
Quantifying speedup (2)



- Say the master takes a [sec] to do the above
- Say the average job granularity is b [sec]
- b / a is the maximum achievable speedup (with arbitrary number of processors)
- a is the reciprocal of the *throughput*



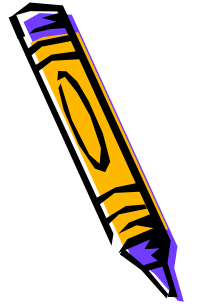
Related Work/SW



- Parallel shells
 - dsh, pdsh, TakTuk [Claudel et al. 2009]
- Overlay scheduler
 - Falkon [Raicu et al. 2007]
- Parallel make
 - Sun dmake, qmake



Ongoing and Future Work



- Real workflow collections (benchmarks)
 - Medline to medie
 - Image stacking of astronomy images
 - Finding supernovae from astronomy images
 - etc.
- Distributed file systems for workflow
- Data-aware schedulers for workflows



Conclusion

- GXP : simple and working software
- Some ideas for “productivity”
 - Smooth transition from “interactive loops” to “scripts/workflow/programs”
 - Don't take risk of job submissions on every single job
 - Separate error-prone steps from real work

