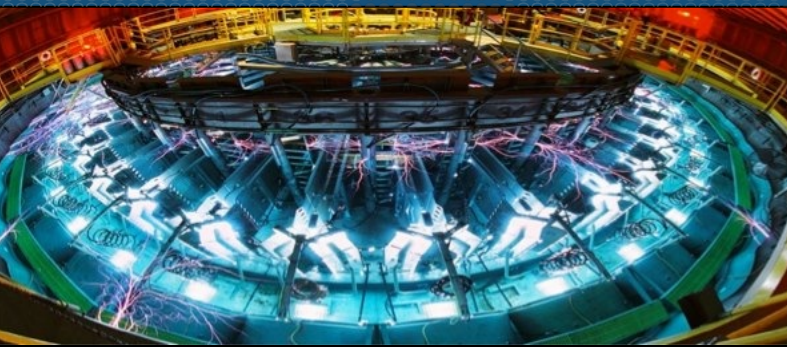
SAND2019-14138C

Exceptional service in the national interest

Sandia National Laboratories



Using Malleable Task Scheduling to Accelerate Package Manager Installations

Samuel Knight*, Jeremiah Wilke*, Todd Gamblin+

*Scalable Modeling and Analysis, Sandia National Labs, Livermore CA

+Lawrence Livermore National Laboratory, Livermore CA

Unclassified Unlimited Release (UUR)

HUST-19, Denver, CO

November 18, 2019

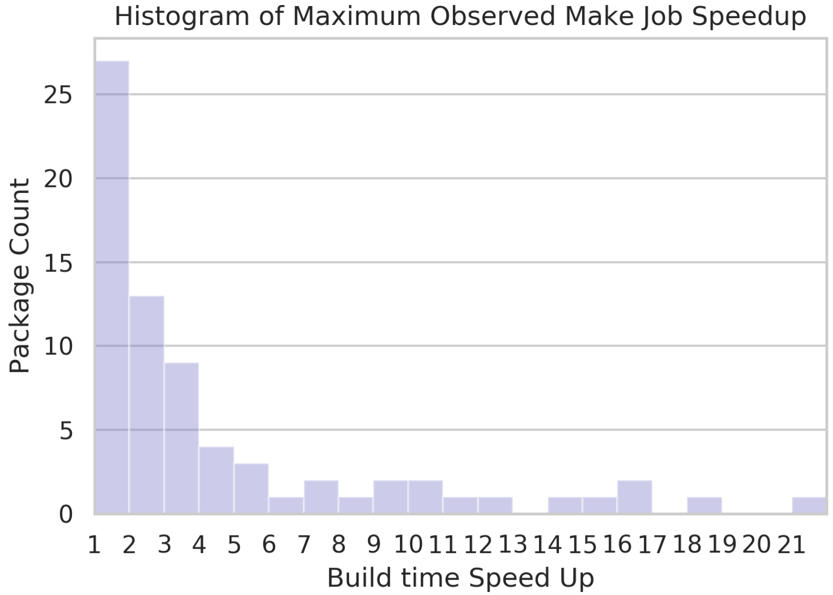
U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

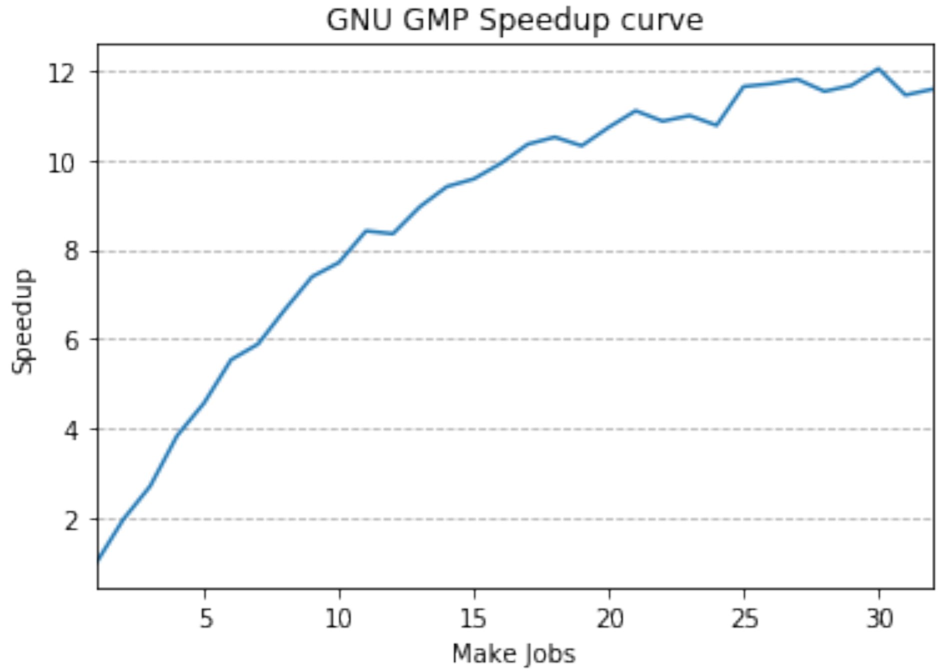
Motivation

- Many system and scientific applications in require compilation from source
- Compilation of large software stacks can be time consuming, e.g. building xSDK can take several hours to build
- Parallel builds can reduce time, but often underutilize hardware resources



Build System Scalability

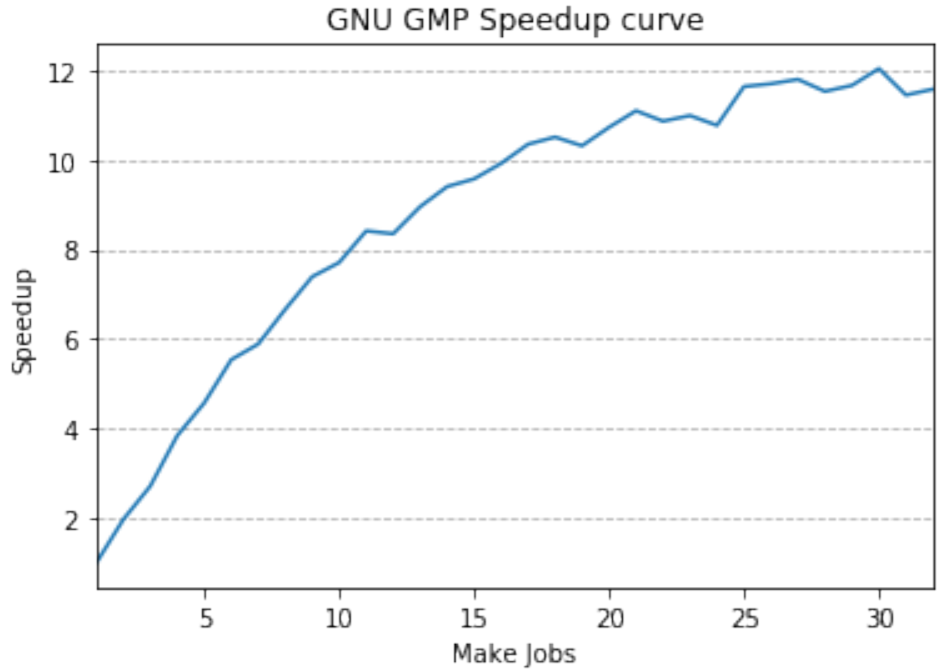
- Build systems tend not to scale linearly as cores are added
- Configure scripts (e.g. Autotools) build and execute many small stub programs serially



Build System Scalability

- Makefiles execute a serial linking step after completing a batch of object files

- Makefiles execute on object files in parallel but sequentially traverse the directory hierarchy, potentially starving itself of jobs at the end of each step.



Spack Manages Installation Dependency Graphs

- Spack installs packages from source code
- Spack analyzes an input Spec and traces the dependencies to create a DAG (Directed Acyclic Graph)
- Concretizer fills in variant, compiler and architecture details not defined by the spec

```
$ spack graph \
> --deptype link,run openmpi
o  openmpi
|\
| |\
| | o  hwloc
| |/|
| | |\
| | o |  libxml2
| |/| |
|/| | |
| | |\ \
o | | | |  zlib
 / / / /
| o | |  xz
|  / /
o | |  numactl
 / /
| o  libpciaccess
|
o  libiconv
```

How Spack Installs Packages

- Spack traces the DAG to find leaf vertices (packages that have no dependencies left to install)

- Select a leaf, install, and repeat until there are no dependencies left in a 'reverse order traversal' of the graph

```
$ spack spec perl@5.30.0~threads
Input spec
--------------------------------
perl@5.30.0~threads

Concretized
--------------------------------
perl@5.30.0%gcc@8.3.1+cpanm+shared~threads arch=linux-rhel8-broadwell
    ^gdbm@1.18.1%gcc@8.3.1 arch=linux-rhel8-broadwell
        ^readline@8.0%gcc@8.3.1 arch=linux-rhel8-broadwell
            ^ncurses@6.1%gcc@8.3.1~symlinks~termlib arch=linux-rhel8-broadwell
                ^pkgconf@1.6.3%gcc@8.3.1 arch=linux-rhel8-broadwell
```

```
[root@75a954ced3c4 /]# spack install perl
==> Installing pkgconf
==> Searching for binary cache of pkgconf
==> Warning: No Spack mirrors are currently co
==> No binary for pkgconf found: installing fr
==> Fetching http://distfiles.dereferenced.org
##############################################
==> Staging archive: /tmp/root/spack-stage/spa
==> Created stage in /tmp/root/spack-stage/spa
==> No patches needed for pkgconf
==> Building pkgconf [AutotoolsPackage]
==> Executing phase: 'autoreconf'
==> Executing phase: 'configure'
==> Executing phase: 'build'
==> Executing phase: 'install'
==> Successfully installed pkgconf
  Fetch: 0.02s.  Build: 9.65s.  Total: 9.68s.
[+] /opt/spack/opt/spack/linux-rhel8-skylake_a
==> Installing ncurses
==> Searching for binary cache of ncurses
==> Warning: No Spack mirrors are currently co
==> No binary for ncurses found: installing fr
==> Fetching http://ftpmirror.gnu.org/ncurses/
##############################################
==> Staging archive: /tmp/root/spack-stage/spa
==> Created stage in /tmp/root/spack-stage/spa
==> No patches needed for ncurses
==> Building ncurses [AutotoolsPackage]
==> Executing phase: 'autoreconf'
==> Executing phase: 'configure'
```

How Spack Installs Packages

- Each package executes a pipeline of phases

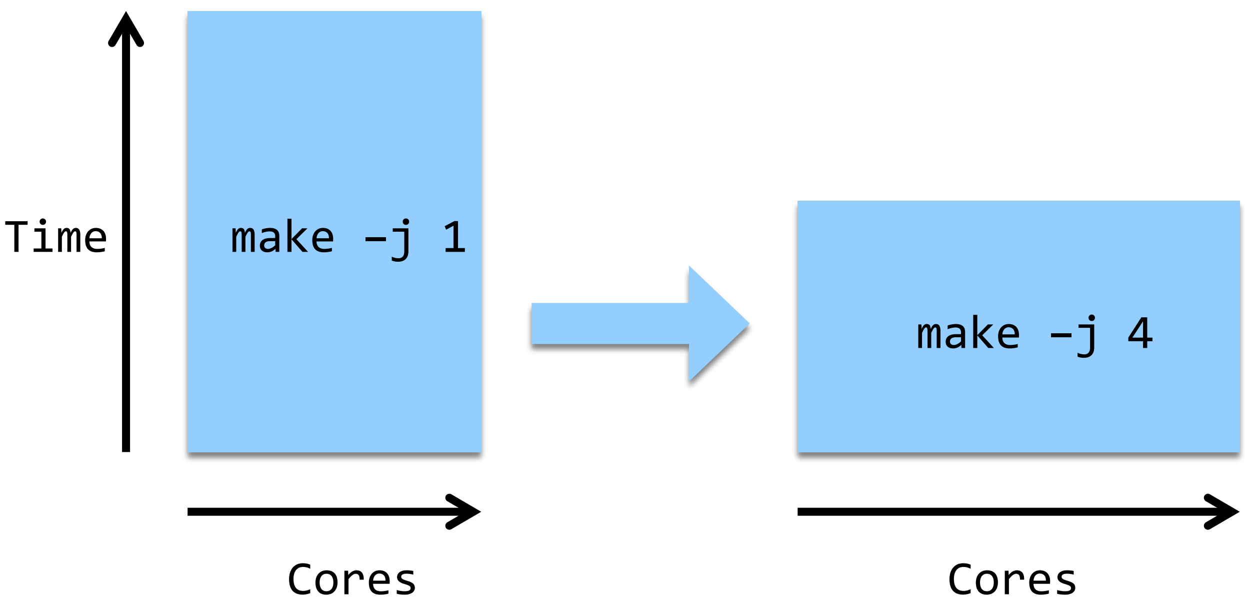
```
==> Building pkgconf [AutotoolsPackage]
==> Executing phase: 'autoreconf'
==> Executing phase: 'configure'
==> Executing phase: 'build'
==> Executing phase: 'install'
==> Successfully installed pkgconf
  Fetch: 0.02s.  Build: 9.65s.  Total: 9.68s.
[+] /opt/spack/opt/spack/linux-rhel8-skylake_a
```

- Phases will implicitly run on every core where possible to exploit intra-task parallelism

```
[root@75a954ced3c4 .spack]# grep -- -j spack-build-out.txt
==> [2019-11-13-16:22:38.379062] 'make' '-j16'
==> [2019-11-13-16:22:40.454473] 'make' '-j16' 'install'
```

Packages are Malleable Tasks in an Installation Graph

- Malleable task – Atomic unit of execution whose completion time changes with the allotment of more resources.

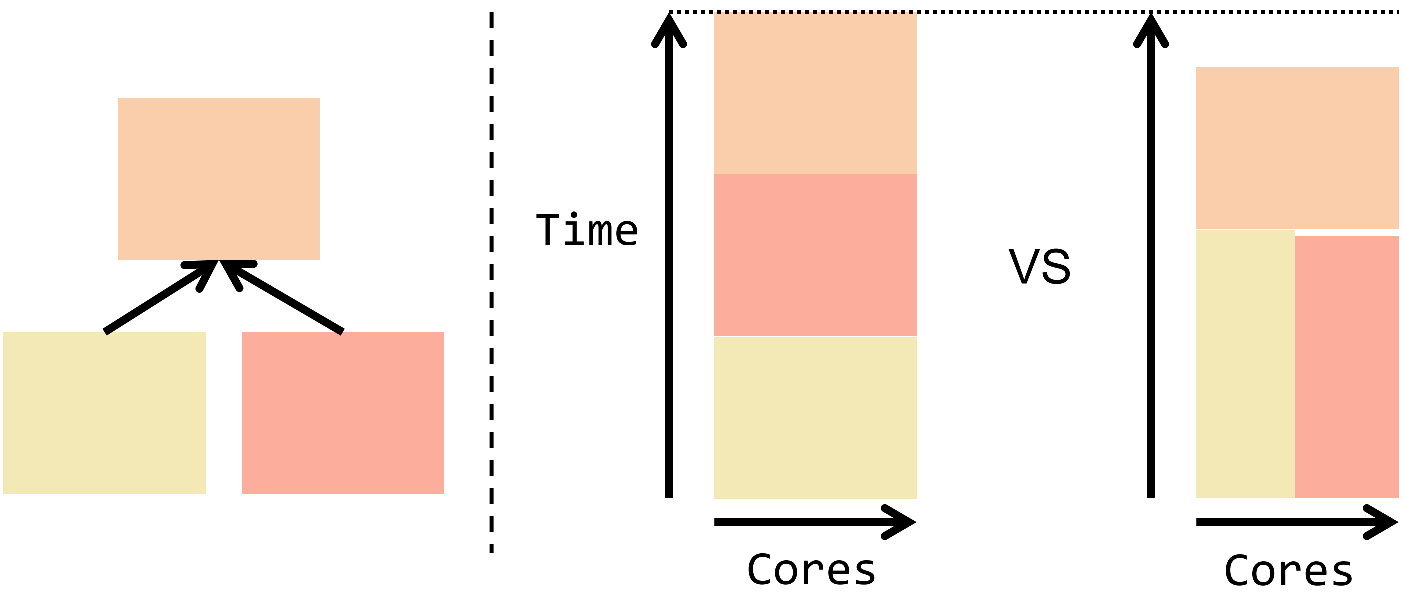



```
$ spack graph \
> --deptype link,run openmpi
o  openmpi
|\
| |\
| | o  hwloc
| |/|
| | |\
| o | |  libxml2
| |/| |
|/| | |
| | |\ \
o | | | |  zlib
 / / / /
| o | |  xz
|  / /
o | |  numactl
 / /
| o  libpciaccess
|
o  libiconv
```

Makefile jobs are an example of malleable tasks

Exploiting Inter-Task Parallelism

- Task DAGs can schedule multiple tasks with a fraction of the available cores or a single task with all cores
- Since build systems do not tend to scale linearly, total installation time can be improved by installing multiple dependencies at the same time.
- Experimental builds will show time improvements >2x

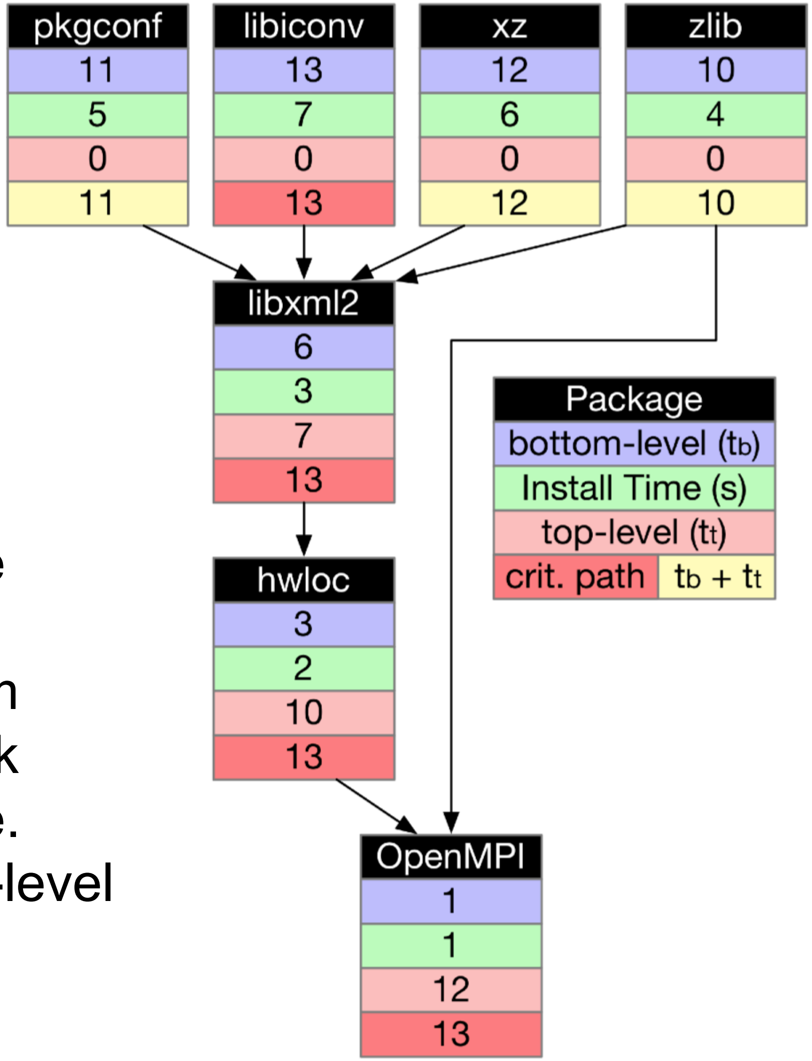


Concepts used in Task Scheduling

- Top Level – the longest path from the current task to an entrance task excluding the task's execution time

- Bottom Level - the longest path by weight from a given task to an exit task including that task's execution time

- Critical Path - the longest path from the current task to an entrance task excluding the task's execution time. Calculated by adding b-level and t-level



Modeling Malleable Task Execution Time

- Ahmdal's law describes an execution as having two parts, a serial and parallelizable component. The serial component results in an upper bound to task speedup

$$t(p) = ((1 - t_{ser})/p + t_{ser})t(1)$$

- If we assume t_{ser} and *t(1)* to be task-intrinsic, the equation can be reduced to

$$t(p) = k_A/p + k_B$$

- k_A and k_B are package-specific constants, determined by measuring execution time with different numbers of cores and fitting with polynomial approximation (least squares)

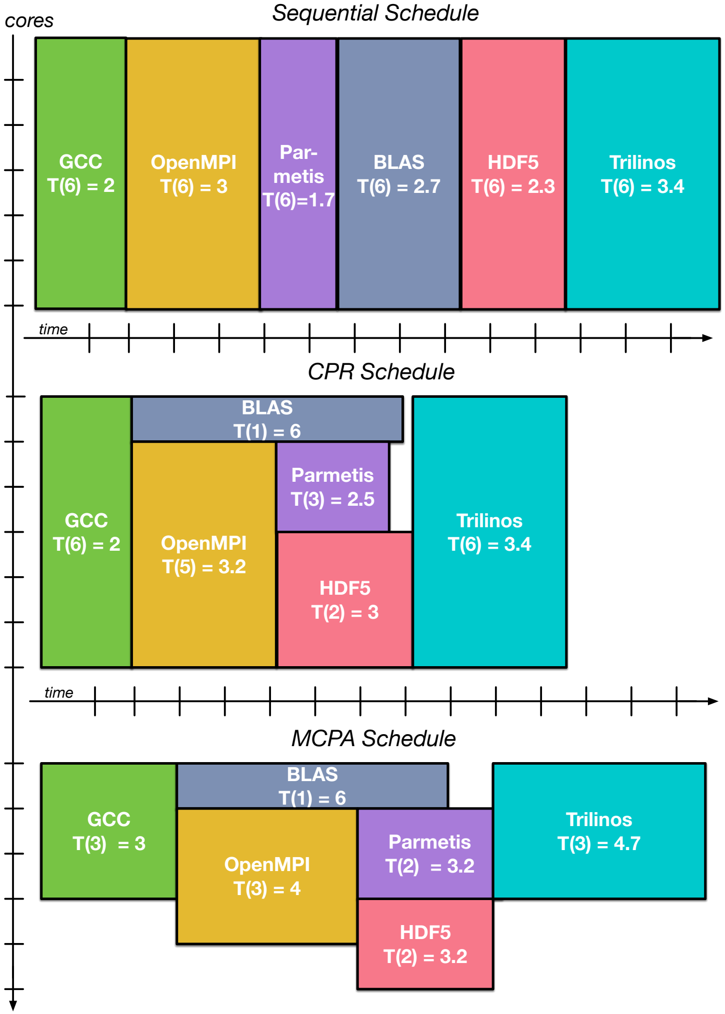
Two-Step Scheduling Algorithms have cost-Benefit Tradeoffs

- M-Task scheduling algorithms are specialized DAG schedulers
- The presented algorithms are "Two-step" schedulers. The first step changes the core allotment, and the second creates a schedule
- CPR (Critical Path Reduction) a greedy algorithm that generally creates good results [1]
- MCPA (Modified Critical Path and Allocation) can yield results similar to CPR at a lower time complexity [2]
- MLS (M-Task List Scheduler) constitutes the second step for both CPR and MCPA, and generates a schedule from a given core allotment

[1] A. Radulescu, C. Nicolescu, A. J. C. van_Gemund and P. P. Jonker, "CPR: mixed task and data parallel scheduling for distributed systems,"

[2] Savina Bansal, Padam Kumar, Kuldip Singh, "An improved two-step algorithm for task and data parallel scheduling in distributed memory machines"

Cost-Benefit Tradeoff



V	Set of vertices
E	Set of edges
P	Set of processors
W	Precedence levels

CPR (Critical Path Reduction) $O(EV^2P + V^3P(logV + PlogP))$
Greedy scheduler that iterates over many possible schedules

MCPA (Modified CPA) $O(V(VW + E)P)$
CPA with additional checks for task parallelism amongst independent tasks

CPR Algorithm

- Every task starts with one core and an initial schedule is created
- Outer loop repeats until the inner loop does not create a better schedule
- Inner loop repeats until it can improve the schedule by adding a core to a task on the critical path or there are no more cores to try

```
procedure CPR(Proc count P, set<Task> tasks)
    for all t_i ∈ tasks do
        p_i ← 1
    end for
    Schedule T ← MLS()
    repeat
        X ← set of tasks where p_i < P
        repeat
            t ← t with max t.t_level + t.b_level
            t.nproc ← t.nproc + 1
            Schedule T′ ← MLS()
            if Length(T′) < Length(T) then
                T ← T′
            else
                t.nproc ← t.nproc − 1
                Remove t from X
            end if
        until T is modified or X is empty
    until T is unmodified
end procedure
```

Scheduler Tweaks

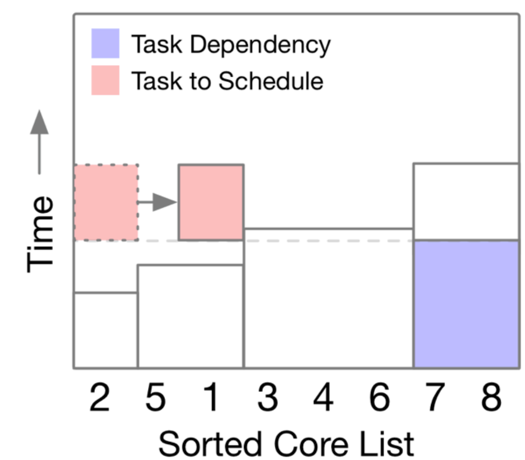
- Filtered CPR (F-CPR) will skip tasks that do not meet a minimum speedup threshold. Value used in experiments was 20% improvement over 8 cores

$$t_{sc} = \begin{cases} Scalable & \text{if } t_c(n)/t_c(1) < threshold, \\ Unscalable & \text{otherwise} \end{cases}$$

- Reuse MLS (R-MLS) leverages memoization between calls from CPR to improve time complexity

- MLS hole filling. The MLS reference inadvertently allows for hole formation, which can be prevented by seeking for processors with later idle times before assigning start times to cores

Partial Schedule



Benchmark Systems

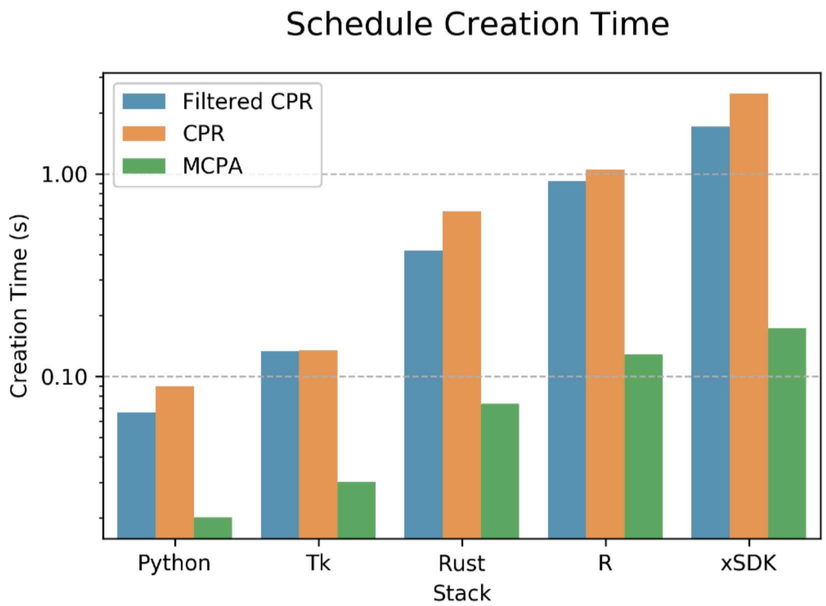
	Node A	Node B
Hardware Cores	32	28
Memory	512 GB	256 GB
Build Mount	SATA III SSD	PCIe NVMe
Install Mount	SATA III SSD	NFS
OS Mount	SATA III SSD	NFS

Benchmarked Packages

Stack	Packages	Phases
Python 2.7.16	14	45
Tk 8.6.8	21	80
Rust 1.33.0	43	149
R 3.5.3	68	248
xSDK 0.4.0	72	222

Schedule Creation Time

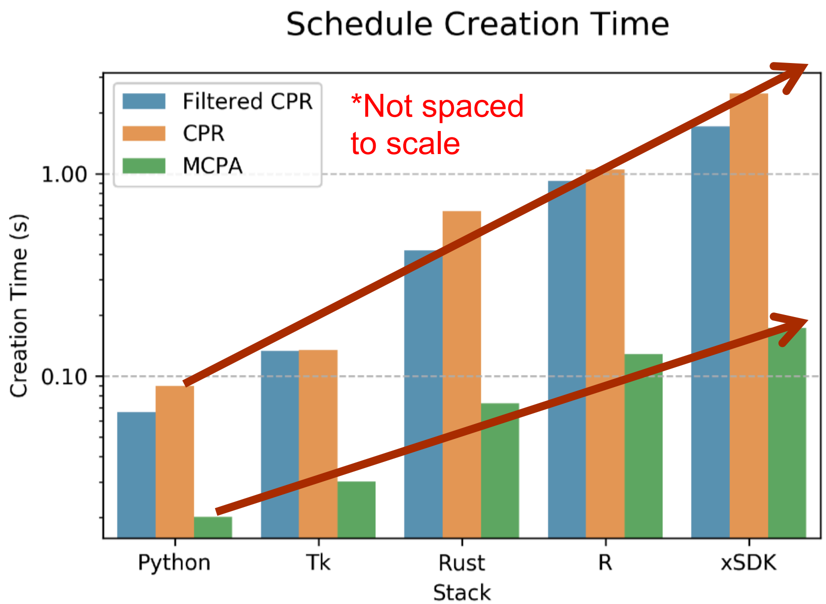
- MCPA had the fastest creation time by order of magnitude
- F-CPR was usually able to create a schedule faster than CPR
- CPR's slowest creation time was 2.49 seconds for a 72 package DAG that installed in 71 minutes



Stack	Execution Time (s)	CPR Creation Time (s)
Python	879.21	0.09
Tk	385.46	0.13
Rust	4563.35	0.65
R	1478.01	1.05
xSDK	4293.00	2.49

Schedule Creation Time

- MCPA had the fastest creation time by order of magnitude
- F-CPR was usually able to create a schedule faster than CPR
- CPR's slowest creation time was 2.49 seconds for a 72 package DAG that installed in 71 minutes

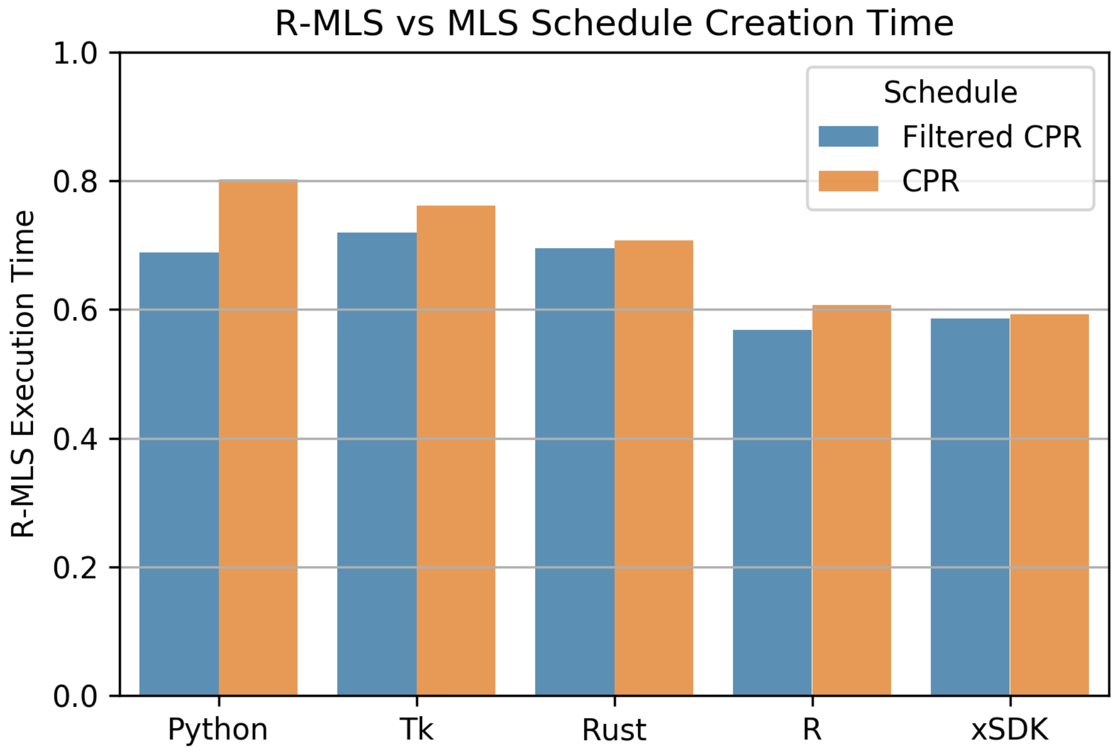


Stack	Execution Time (s)	CPR Creation Time (s)
Python	879.21	0.09
Tk	385.46	0.13
Rust	4563.35	0.65
R	1478.01	1.05
xSDK	4293.00	2.49

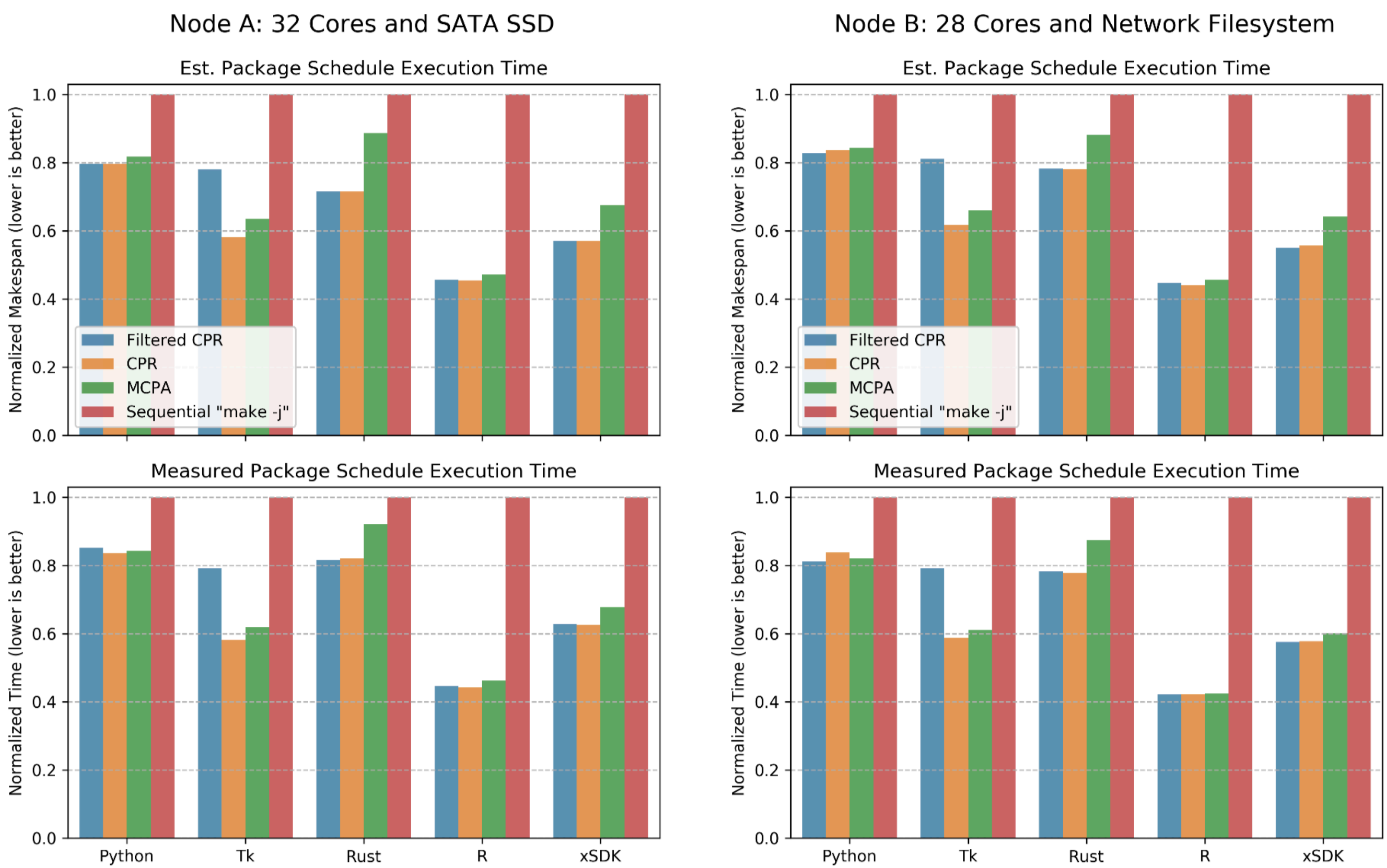
5.1x packages 27.7x creation time

MLS vs R-MLS Schedule Creation Time

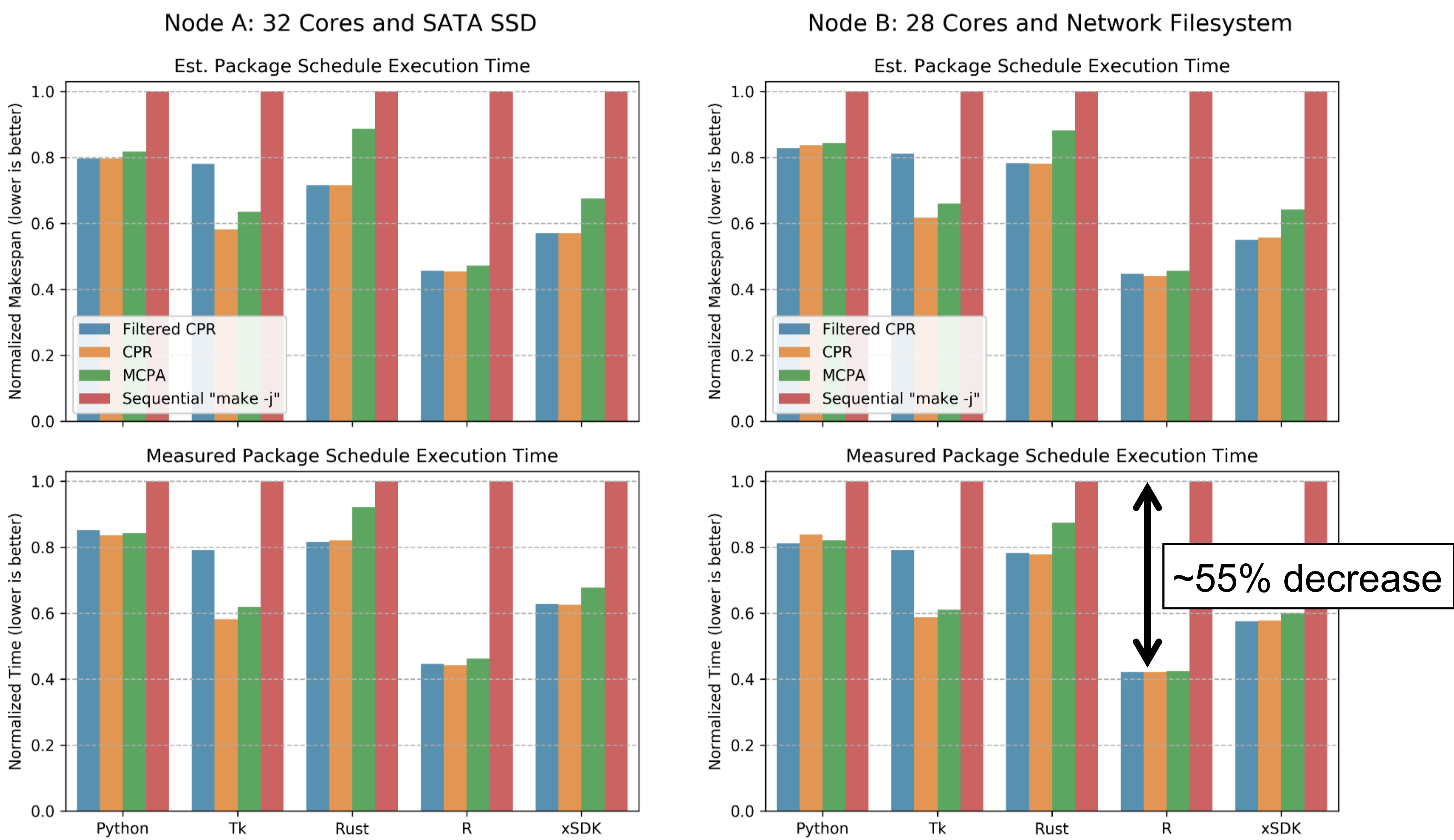
- [F-]CPR schedules saw up to 42% creation time improvement



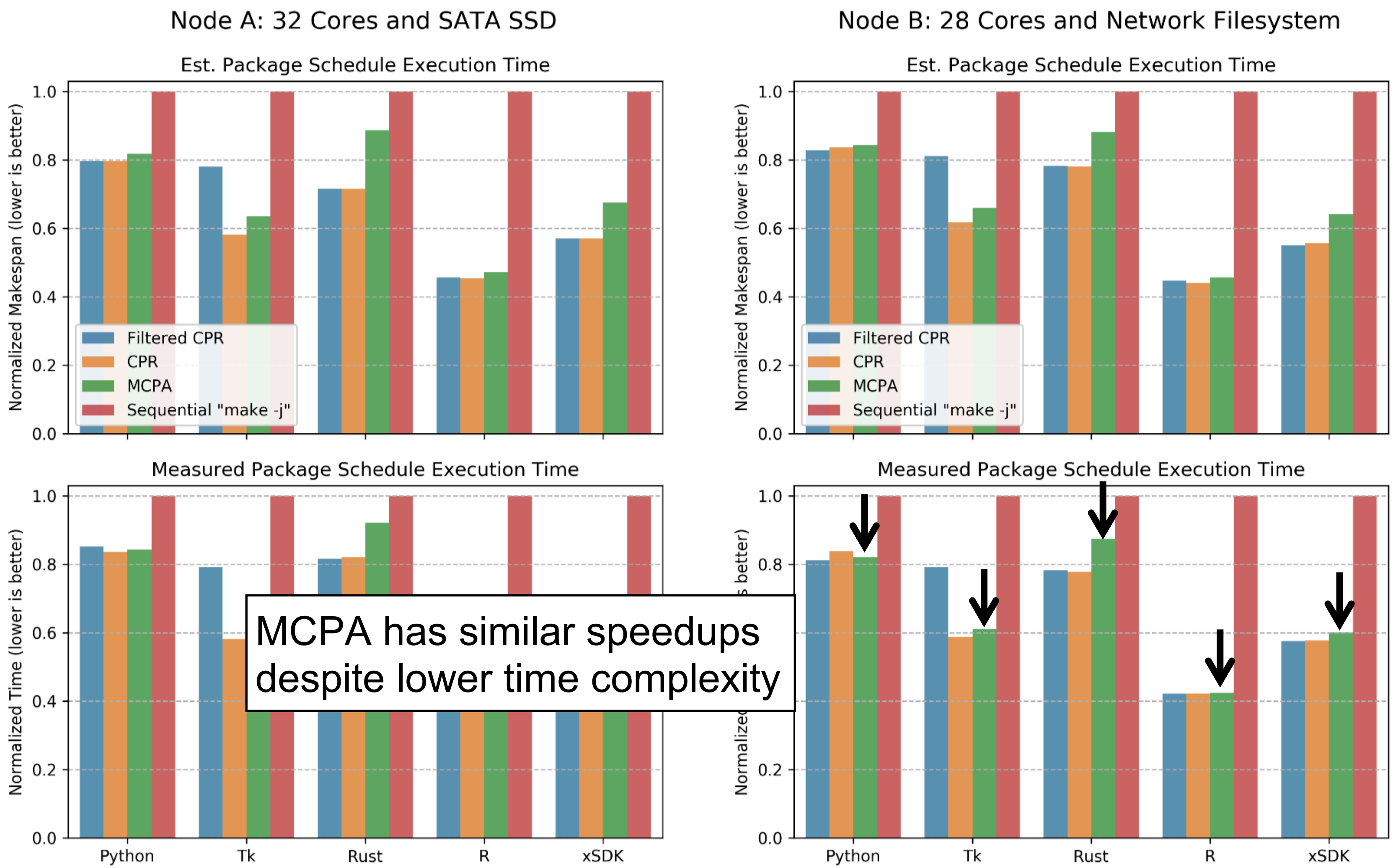
Schedule Execution time



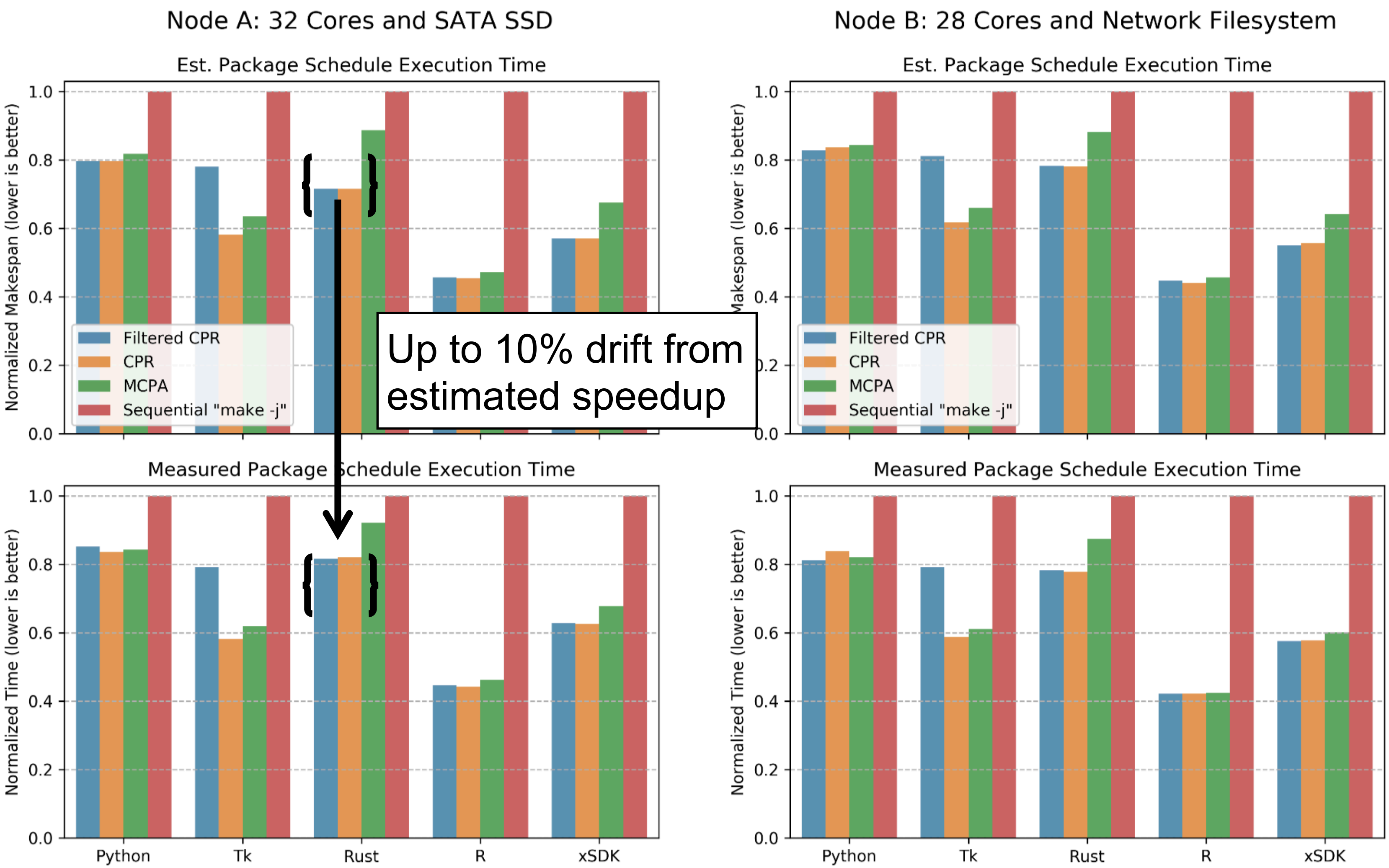
Schedule Execution time



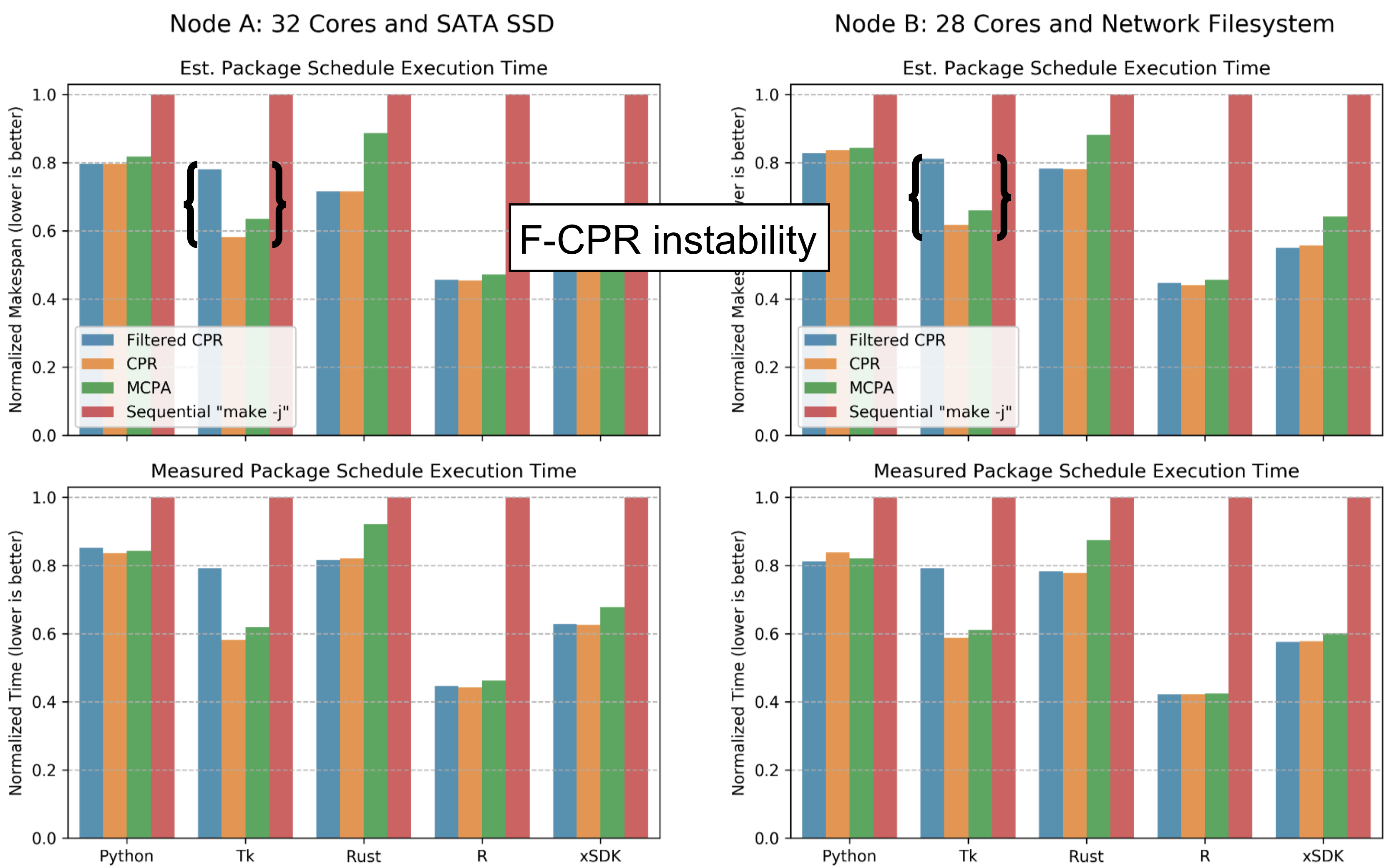
Schedule Execution time



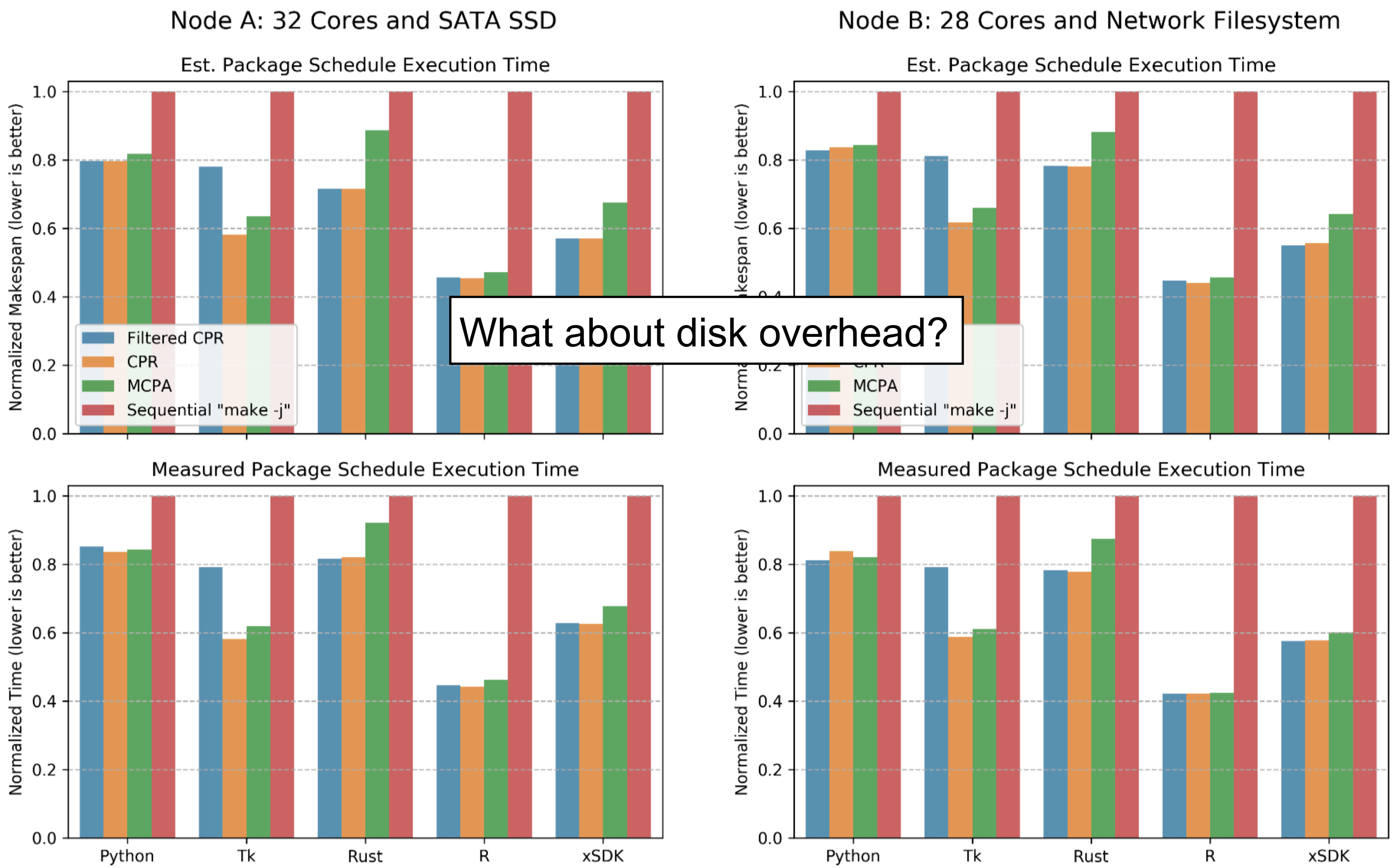
Schedule Execution time



Schedule Execution time

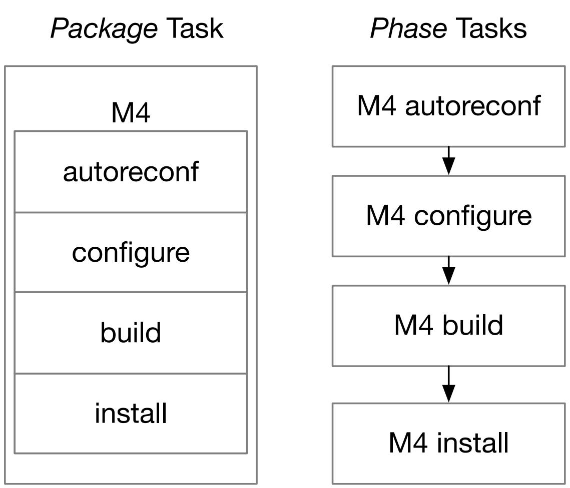


Schedule Execution time



Future Work

- Multi-node MLS – MLS can be trivially changed to create schedules for multiple nodes
- Task execution time heuristic has a primitive implementation. It is not portable across machines, and was not designed to take into account package details like variants
- Hyperthreading and overprovisioning may provide more time reduction
- Phase tasks instead of package task DAGs
- The scheduler has no model to account for package fetching



Conclusion

Sandia National Laboratories

- Software installation does not tend to utilize system resources optimally
- Package execution times can be modeled as malleable tasks and organized with a DAG scheduler
- In every tested case, the schedulers took an insignificant amount of time to produce a greatly improved installation times over a sequentially installed stack
- Code: https://github.com/sknigh/spack/tree/feature/parallelbuild4

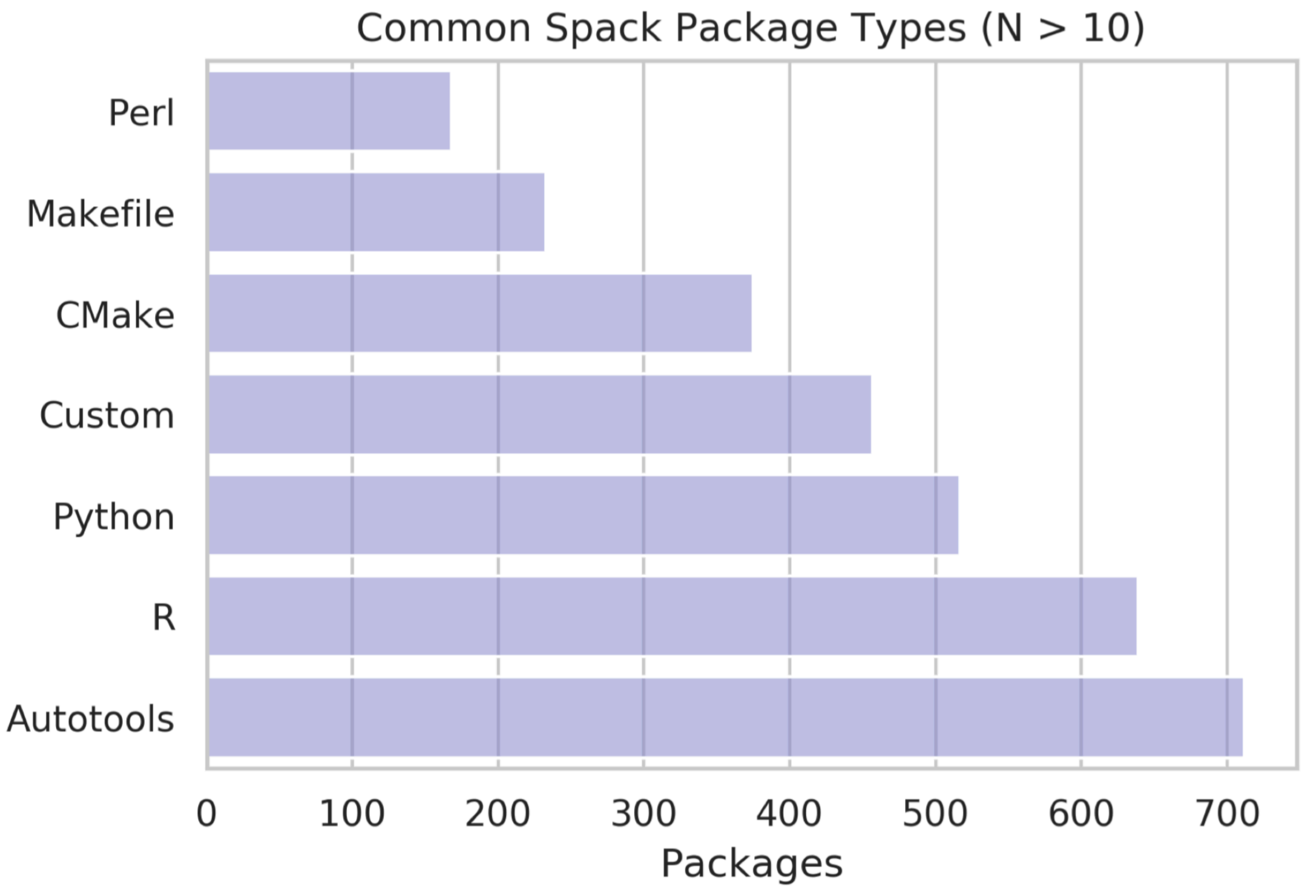
Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Table of Symbols

Symbol	Definition
T	Set of all tasks
t_i	ith task in set of all tasks
$t_i(p_i)$	Execution time of ith task with p processors
p_i	No. processors allotted to ith task
w_i	ith Task work area $(t_i(p) \times p)$
t_s	Start time of ith task
t_f	Finish time of ith task
t_{ser}	Serial proportion of task execution
t_{sc}	Whether a task is scalable
t_t	Task top-level
t_b	Task bottom-level
V	Set of vertices
E	Set of edges
P	Set of processors
W	Precedence levels

Spack Package Composition



M-Task List Scheduling (MLS) Algorithm

- Takes a list of tasks with cores already allotted
- Sorts the tasks by b-level (e.g. how deep they are in the dependency hierarchy)
- "Assign" the task by adding its execution time to the cores with the earliest idle times
- Assign the earliest idle time to the task

```
procedure MLS(Proc count P, set<Task> tasks, set<Core> cores)
    tasks ← sort tasks by b-level
    for all c ∈ cores do
        c.idle_time ← 0
    end for
    for all t ∈ tasks do
        sortedCores ← sorted cores by idle time
        selectedCores ← sortedCores[0 : p(t)]
        offset ← 0
        t.start_time ← latest selectedCores or dependency end
time
        t.end_time ← t.start_time + t.exec_time
        while sortedCores[p(t)+ offset +1] ≤ t.start_time do
            offset ←offset+1
        end while
        for i ←offset, p(t)+offset do
            sortedCores[i] ← t.end_time
        end for
    end for
end procedure
```

MCPA Algorithm

- MCPA finds a task that minimizes 'Work Area' gain along the critical path
- Will not allocate cores to tasks when the total cores allocated the the precedence level are equivalent to the total cores
- Stops when critical path work area is greater than global average work area

```
procedure MCPA(In: Proc count P, In-Out: set<Task> tasks)
    for all t ∈ tasks do
        t.ncores = 1
    end for
    computeTandBLevels(tasks)
    while L_cp > A_p do
        CP ← set of tasks on current critical path
        ValidT ← ∅
        for all t ∈ CP do
            if cores available at t's precedence level then
                ValidT ← t
            end if
        end for
        t_opt ← bestWorkArea(ValidT)
        t_opt.ncores ← t_opt.ncores + 1
        computeTandBLevels(tasks)
    end while
end procedure
procedure BESTWORKAREA(set<Task> tasks)
    t_opt ← NULL
    G_opt ← inf
    for all t_i ∈ tasks do find max work area gain G
        G_i ← w_i(n_i)/n_i − w_i(n_i+1)/(n_i+1)
        if G_i > G_opt then
            t_opt ← t_i
            G_opt ← G_i
        end if
    end for
end procedure
```

Algorithm Descriptions

Algorithm	Complexity	Description
CPR (Critical Path Reduction)	$O(EV^2P + V^3P(logV + PlogP))$	Greedy scheduler that iterates over many possible schedules
F-CPR (Filtered CPR)	$O(EV^2P + V^3P(logV + PlogP))$	CPR with minimum improvement threshold ("filter") for pruning search space
CPA (Critical Path and Allocation)	$O(V(V + E)P)$	Allots cores on critical path until it reaches average processor area
MCPA (Modified CPA)	$O(V(VW + E)P)$	CPA with additional checks for task parallelism amongst independent tasks
MLS (M-task List Scheduling)	O(E + Vlog(V) + VPlogP)	Basic scheduling algorithm for assigning task start times.
R-MLS (Reuse MLS)	$O(E + VPlogP)$	MLS with memoization to reduce CPR's time complexity