Parallel Partition Backtrack

Jason B. Hill

University of Colorado at Boulder

5th de Brún Workshop, Galway, April 2011

Table of Contents

Introduction

- This talk is more about practice than theory. The purpose is simply to convince one that backtrack searches in permutation groups can be helped by parallel processing (this is not at all obvious). Mine may not be the best approach, but it is a start.
- Several references for theory are in upcoming slides.
- For partition backtrack (although in different notation), see:

  Leon, Jeffrey S., "Partitions, refinements, and permutation group computation," Dimacs Series in Discrete Mathematics and Theoretical Computer Science, vol 28 (1997), 123-158.

- For the code, examples and explanations (coming):

  `http://math.jasonbhill.com/backtrack`

Introduction

- Since January between math, applied math and CS departments at Colorado: 8 seminar talks containing “GPU” in the title.

Introduction

- Since January between math, applied math and CS departments at Colorado: 8 seminar talks containing "GPU" in the title.
- One number theory student told me: "I don't use the open-source options because `Mathematica` can use multiple cores and my GPU."

Introduction

- Since January between math, applied math and CS departments at Colorado: 8 seminar talks containing “GPU” in the title.
- One number theory student told me: “I don’t use the open-source options because `Mathematica` can use multiple cores and my GPU.”
- My reply: “A single horse may pull a cart perfectly well. Sometimes, not always, two horses can do a better job. It is rare (but perhaps possible) that using 2,000 chickens could improve that situation.”

Introduction

- Since January between math, applied math and CS departments at Colorado: 8 seminar talks containing "GPU" in the title.
- One number theory student told me: "I don't use the open-source options because `Mathematica` can use multiple cores and my GPU."
- My reply: "A single horse may pull a cart perfectly well. Sometimes, not always, two horses can do a better job. It is rare (but perhaps possible) that using 2,000 chickens could improve that situation."
- During an algebra seminar on non-polynomial time permutation group algorithms, someone asked me why backtrack searches in groups are not performed in parallel.

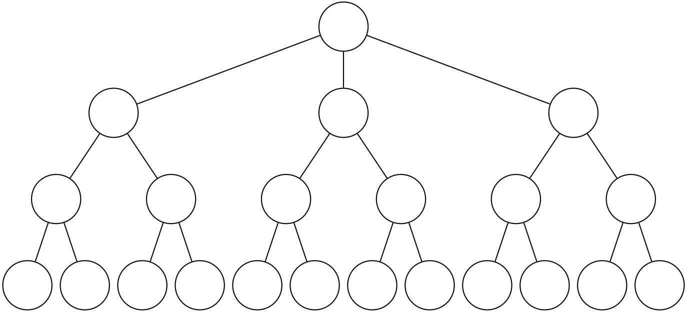
Partition Backtrack

The Very Basics of Backtrack

- The term “backtrack” was coined by D.H. Lehmer in the 1950s.

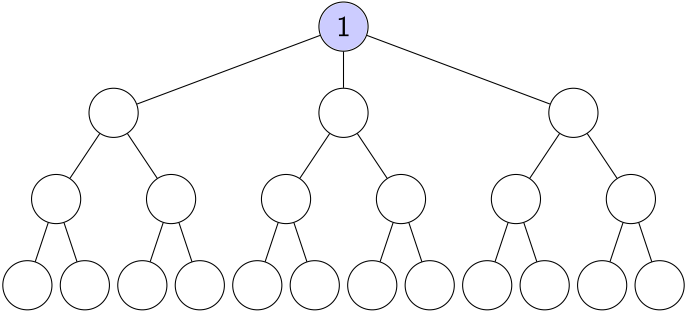
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



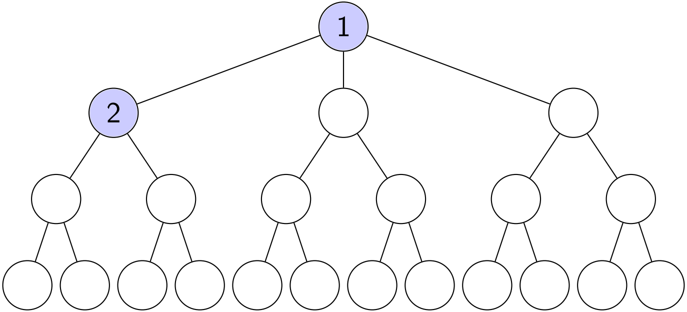
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



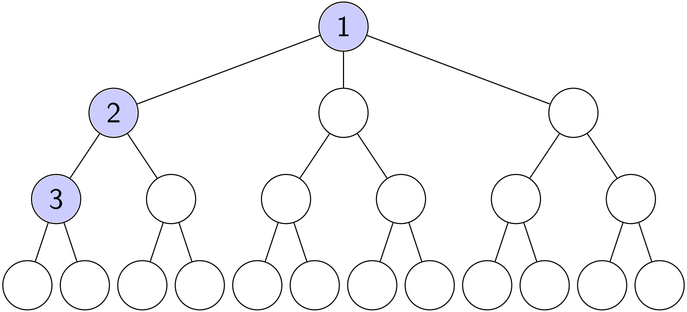
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



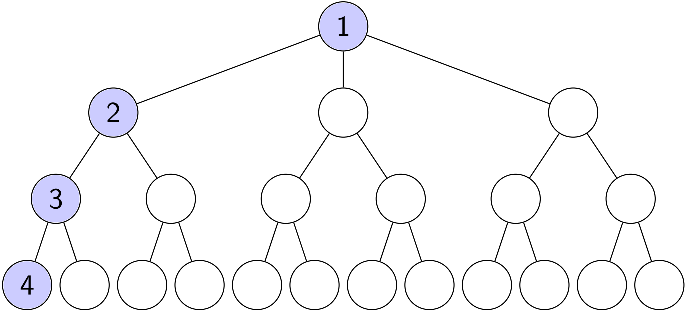
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



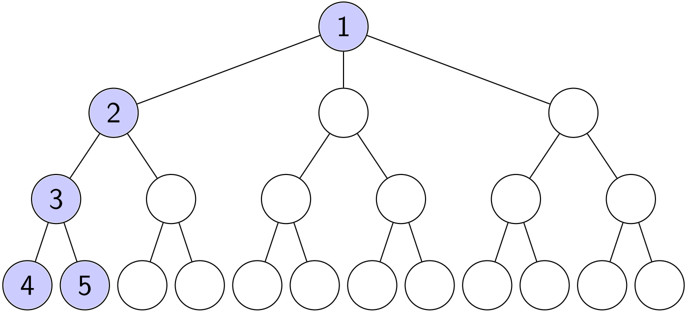
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



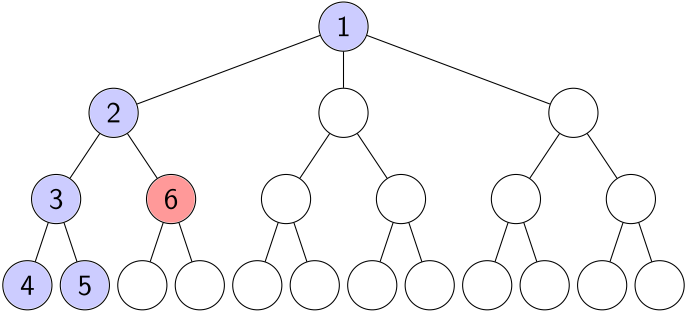
The Very Basics of Backtrack

- The term “backtrack” was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a “depth first search” for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



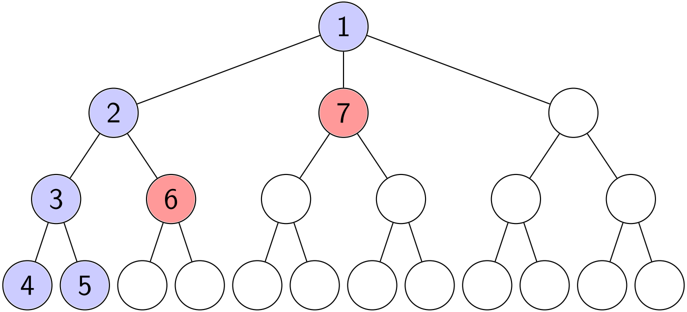
The Very Basics of Backtrack

- The term “backtrack” was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a “depth first search” for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



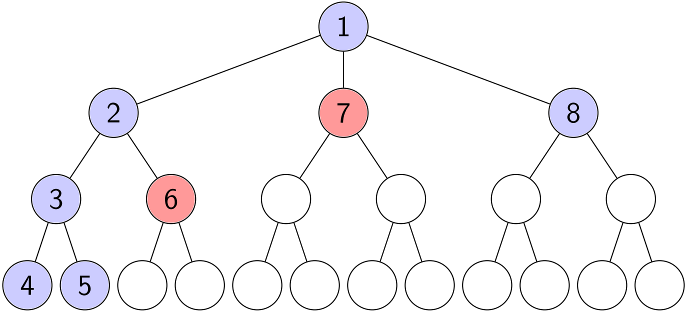
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



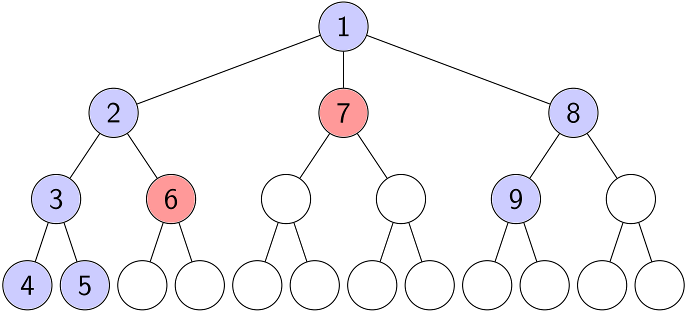
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



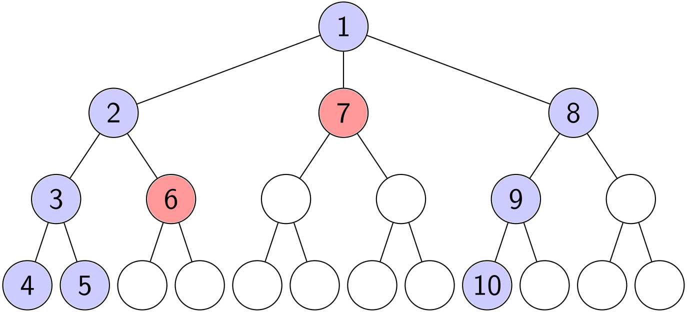
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



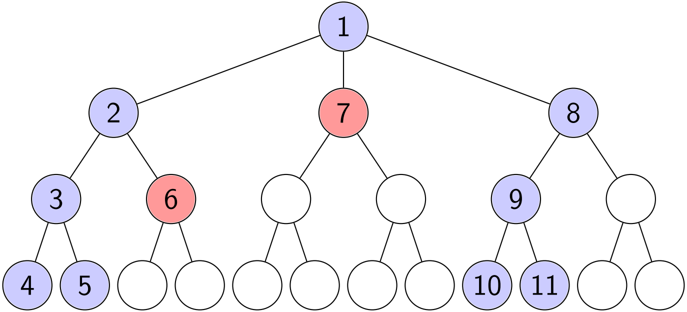
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



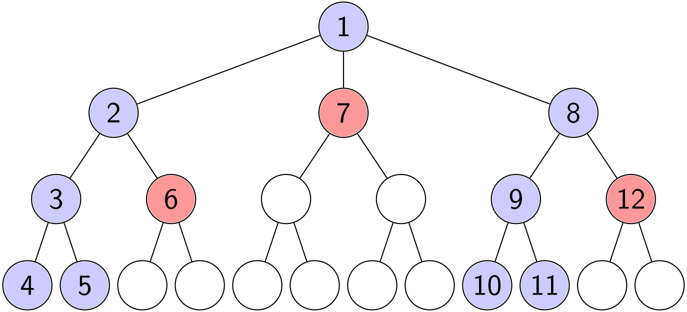
The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



The Very Basics of Backtrack

- The term "backtrack" was coined by D.H. Lehmer in the 1950s.
- Main Idea: Traverse a search tree recursively from the root down using a "depth first search" for leaves at the bottom of the tree that may satisfy some property. Never search below a node if it becomes evident that no leaf below that node will satisfy the property.



Backtrack in Permutation Groups

- We use backtrack searches for many problems where no polynomial runtime algorithm is known to exist. (e.g., normalizers, centralizers, subset stabilizers).

Backtrack in Permutation Groups

- We use backtrack searches for many problems where no polynomial runtime algorithm is known to exist. (e.g., normalizers, centralizers, subset stabilizers).
- The basic approach builds on the work of Sims (1971). See:
  - Alexander Hulpke's online CGT notes
  - Permutation Group Algorithms by Ákos Seress
  - Handbook of CGT by Holt, Eick, and O'Brien

Backtrack in Permutation Groups

- We use backtrack searches for many problems where no polynomial runtime algorithm is known to exist. (e.g., normalizers, centralizers, subset stabilizers).
- The basic approach builds on the work of Sims (1971). See:
  - Alexander Hulpke's online CGT notes
  - Permutation Group Algorithms by Ákos Seress
  - Handbook of CGT by Holt, Eick, and O'Brien
- In 1981, B. McKay used "partition backtrack" in his program *Nauty* for testing graph isomorphisms.

Backtrack in Permutation Groups

- We use backtrack searches for many problems where no polynomial runtime algorithm is known to exist. (e.g., normalizers, centralizers, subset stabilizers).
- The basic approach builds on the work of Sims (1971). See:
  - Alexander Hulpke's online CGT notes
  - Permutation Group Algorithms by Ákos Seress
  - Handbook of CGT by Holt, Eick, and O'Brien
- In 1981, B. McKay used "partition backtrack" in his program *Nauty* for testing graph isomorphisms.
- In 1991 and 1997, J. Leon published papers detailing partition backtrack in permutation groups. He wrote an implementation in C between those papers.

Backtrack in Permutation Groups

- We use backtrack searches for many problems where no polynomial runtime algorithm is known to exist. (e.g., normalizers, centralizers, subset stabilizers).
- The basic approach builds on the work of Sims (1971). See:
  - Alexander Hulpke's online CGT notes
  - Permutation Group Algorithms by Ákos Seress
  - Handbook of CGT by Holt, Eick, and O'Brien
- In 1981, B. McKay used "partition backtrack" in his program *Nauty* for testing graph isomorphisms.
- In 1991 and 1997, J. Leon published papers detailing partition backtrack in permutation groups. He wrote an implementation in C between those papers.
- GAP and Magma use partition backtrack.

Partition Backtrack

Definitions and Notation

Definition A **partition** λ of $n \in \mathbb{Z}$ is a set composition of disjoint non-empty subsets of $\{1, 2, \ldots, n\}$. Then λ_i denotes the ith subset of λ.

Example For $n = 7$, one example is $\lambda = [\{5\}, \{1, 3, 4\}, \{2, 7\}, \{6\}]$. We will view partitions as tableau-like diagrams:

$$\lambda = \begin{array}{|c|c|c|} \hline 5 \\ \cline{1-1} 1 & 3 & 4 \\ \hline 2 & 7 \\ \cline{1-2} 6 \\ \cline{1-1} \end{array} = \begin{array}{|c|c|c|} \hline 5 \\ \cline{1-1} 3 & 1 & 4 \\ \hline 2 & 7 \\ \cline{1-2} 6 \\ \cline{1-1} \end{array} .$$

Here, $|\lambda_1| = |\lambda_4| = 1$, $|\lambda_2| = 3$ and $|\lambda_3| = 2$.

Definition The **height** $h(\lambda)$ of a partition λ is the number of non-empty rows. In the example above, $h(\lambda) = 4$.

Partition Backtrack

Definitions and Notation

Definition Given two partitions λ and μ of n, μ is a **refinement** of λ, written $\mu \leq \lambda$, if for $1 \leq i \leq h(\lambda)$ we have $\mu_i \subseteq \lambda_i$.

Example

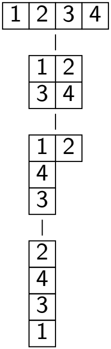
$$\begin{array}{|c|c|} \hline 5 & \\ \hline 3 & \\ \hline 2 & 7 \\ \hline 6 & \\ \hline 1 & 4 \\ \hline \end{array} \leq \begin{array}{|c|c|c|} \hline 5 & & \\ \hline 3 & 1 & 4 \\ \hline 2 & 7 & \\ \hline 6 & & \\ \hline \end{array} .$$

Definition A **partition stack** $\underline{\lambda}$ is a sequence of partitions λ of n satisfying the property that the kth partition is a refinement of the $(k-1)$th partition.

Definition A **complete** partition stack $\underline{\lambda}$ is a stack with the property that the kth partition λ satisfies $h(\lambda) = k$.

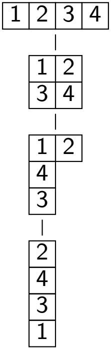
Partition Backtrack

Example of a complete partition stack (Refinements proceed downwardly.)



Partition Backtrack

Example of a complete partition stack (Refinements proceed downwardly.)



Note that recording this complete partition stack may be done efficiently by recording only the row added at each refinement: $[\{3,4\},\{3\},\{1\}]$

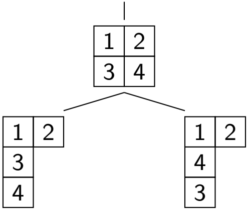
Partition Backtrack

Main Idea: (very briefly – skipping massive details)

- Let G be a group acting on domain $\Omega = \{1, \ldots, n\}$ with base B. We wish to find elements of G satisfying some property $\mathcal{P}$.
- Start with a partition λ of n having height 1.
- Refine λ in a complete partition stack, exploiting the property $\mathcal{P}$ as much as possible to perform the refinements.
- When $\mathcal{P}$ provides no refinement, refine rows of λ containing a base element by mapping that base element to other integers in that row.

Partition Backtrack

- For example, if no refinement from $\mathcal{P}$ is known and we are currently considering $\lambda = [\{1,2\},\{3,4\}]$ with 4 a base point, then we have either $4 \mapsto 4$ or $4 \mapsto 3$.



- We backtrack using the possible base images, never constructing refinements below a given partition if we determine that no group element satisfying $\mathcal{P}$ can exist below that node.

Partition Backtrack

- Leon's C code for partition backtrack can be found in the GAP package GUAVA. It can perform partition backtrack efficiently (should you know how to use it) on (among other problems):
  - set stabilizers
  - partition stabilizers
  - element and subgroup centralizers
  - isomorphisms and automorphism groups of designs
  - isomorphisms and automorphisms of linear codes

Partition Backtrack

- David Joyner convinced Leon to GPL his code in 2007. Robert Miller and Tom Boothby revised the code.
- Robert Miller is currently working on a Cython implementation for Sage.
- I've used Leon's code as a launchpad, modernizing the C and adding MPI (and hopefully OpenMP soon) routines.

Introduction to Parallelism

Introduction to Parallelism

Introduction to Parallelism

- 10 years ago, if you wanted to run a program faster, one option was to simply wait a year or two. Faster processors would be available.

Introduction to Parallelism

Introduction to Parallelism

- 10 years ago, if you wanted to run a program faster, one option was to simply wait a year or two. Faster processors would be available.
- In 1965, Intel co-founder G.E. Moore published an article in *Electronics Magazine* noting that circuits were consistently doubling in complexity roughly ever 2 years. He predicted this trend would continue for at least 10 years.

Introduction to Parallelism

Introduction to Parallelism

- 10 years ago, if you wanted to run a program faster, one option was to simply wait a year or two. Faster processors would be available.
- In 1965, Intel co-founder G.E. Moore published an article in *Electronics Magazine* noting that circuits were consistently doubling in complexity roughly ever 2 years. He predicted this trend would continue for at least 10 years.
- His prediction was **very** accurate until around 2004.

Introduction to Parallelism

Introduction to Parallelism

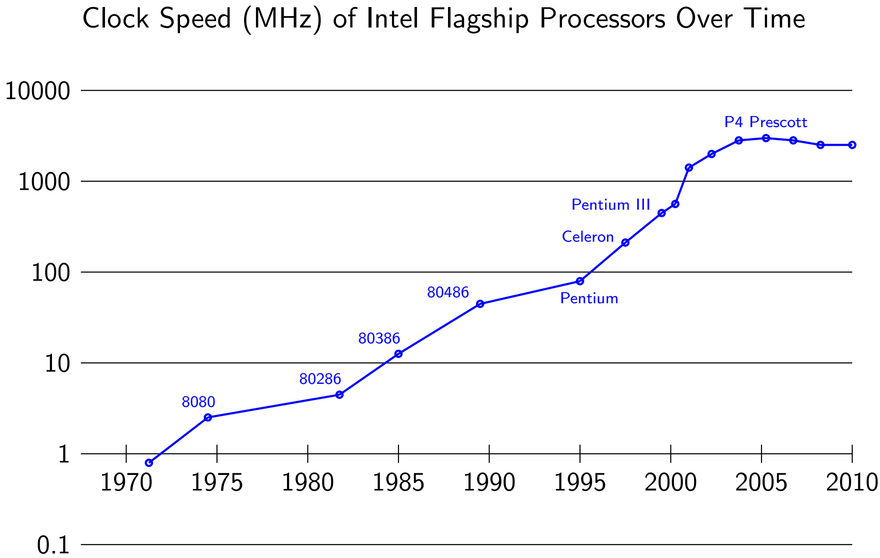
- 10 years ago, if you wanted to run a program faster, one option was to simply wait a year or two. Faster processors would be available.
- In 1965, Intel co-founder G.E. Moore published an article in *Electronics Magazine* noting that circuits were consistently doubling in complexity roughly ever 2 years. He predicted this trend would continue for at least 10 years.
- His prediction was **very** accurate until around 2004.
- After 2004, the clock speed of commercially available CPUs actually decreased. 4 GHz CPUs are simply too expensive to cool.

Introduction to Parallelism

Introduction to Parallelism

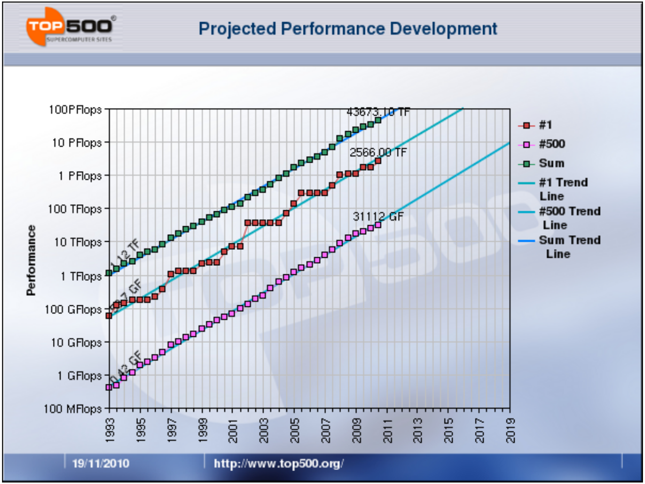
- 10 years ago, if you wanted to run a program faster, one option was to simply wait a year or two. Faster processors would be available.
- In 1965, Intel co-founder G.E. Moore published an article in *Electronics Magazine* noting that circuits were consistently doubling in complexity roughly ever 2 years. He predicted this trend would continue for at least 10 years.
- His prediction was **very** accurate until around 2004.
- After 2004, the clock speed of commercially available CPUs actually decreased. 4 GHz CPUs are simply too expensive to cool.
- This isn't a “I'll wait another year or two and it will be fixed” sort of problem. It's the laws of physics. CPUs will not get faster.

Introduction to Parallelism



Introduction to Parallelism

- At the same time, multi-core systems have become the norm. Ironically, the number of flops (floating point operations per second) computers are capable of is still increasing at the same rate as before.



Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.

Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.
- Amdahl's Law provides bounds for the speed-up of partially parallelizable algorithms. The results are, usually, not great.

Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.
- Amdahl's Law provides bounds for the speed-up of partially parallelizable algorithms. The results are, usually, not great.
- Communication between processors/cores is drastically more expensive than computation is, and varies by hardware.

Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.
- Amdahl's Law provides bounds for the speed-up of partially parallelizable algorithms. The results are, usually, not great.
- Communication between processors/cores is drastically more expensive than computation is, and varies by hardware.
- Communications can create deadlock situations.

Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.
- Amdahl's Law provides bounds for the speed-up of partially parallelizable algorithms. The results are, usually, not great.
- Communication between processors/cores is drastically more expensive than computation is, and varies by hardware.
- Communications can create deadlock situations.
- Splitting fast-running algorithms (even complex and parallelizable ones!!) across many cores may result in a communication cost that causes the program to run slower than it runs in serial (single-core).

Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.
- Amdahl's Law provides bounds for the speed-up of partially parallelizable algorithms. The results are, usually, not great.
- Communication between processors/cores is drastically more expensive than computation is, and varies by hardware.
- Communications can create deadlock situations.
- Splitting fast-running algorithms (even complex and parallelizable ones!!) across many cores may result in a communication cost that causes the program to run slower than it runs in serial (single-core).
- Memory management is much more challenging.

Introduction to Parallelism

There are big problems:

- Most algorithms are only partially parallelizable.
- Amdahl's Law provides bounds for the speed-up of partially parallelizable algorithms. The results are, usually, not great.
- Communication between processors/cores is drastically more expensive than computation is, and varies by hardware.
- Communications can create deadlock situations.
- Splitting fast-running algorithms (even complex and parallelizable ones!!) across many cores may result in a communication cost that causes the program to run slower than it runs in serial (single-core).
- Memory management is much more challenging.
- The whole area suffers from lack of standardization: OpenMP, OpenMPI, MPICH, LAM/MPI, pyMPI, CUDA, OpenMPC, etc.

An Implementation

An Implementation

Goal: Write a parallel partition backtrack program that...

- ... gets correct results (obvious requirement).
- ... can actually be used (i.e., called from GAP or terminal).
- ... uses knowledge of communications -vs- computation costs specific to the host machine at runtime to determine how to parallelize.
- ... does not slow down relative to the serial version.

This is a work in progress, but it does actually work.

- Uses Leon's code as a starting point.
- Currently implemented for centralizers and set stabilizers.
- Currently input/output is in the terminal with files (very inefficient). Plan to add process/stream capabilities from GAP.

An Implementation

Test Platforms: Commodity Computers

- The code (when completed) should be callable from GAP.
- At present, the code works on these machines through terminal commands and using files for input and output.

Machine	CPU	cores	RAM	Gflops
Dirichlet	1x U3500 @ 1.4 GHz	1	4 GB	3
Descartes	1x E5200 @ 2.5 GHz	2	4 GB	13
Tarski	1x Q9400 @ 2.6 GHz	4	8 GB	38
Euclid	2x Xeon E5440 @ 2.8 GHz	8	24 GB	49
Sage	2x Opteron 6128 @ 2.0 GHz	16	24 GB	63

An Implementation

Test Platforms: Supercomputers

- The code is callable from a scheduler process on these machines.

NSF TeraGrid (Boulder, Pittsburgh, San Diego)

Machine	CPU	cores	RAM	Gflops
Frost	4096x PPC-440 @ 700 MHz	8192	2 TB	22936
Blacklight	512x Xeon X7560 @ 2.3 GHz	4096	32 TB	36864
Trestles	1296x Opteron 6136 @ 2.4 GHz	10368	21 TB	100000

An Implementation

Test Platforms: Supercomputers

- The code is callable from a scheduler process on these machines.

NSF TeraGrid (Boulder, Pittsburgh, San Diego)

Machine	CPU	cores	RAM	Gflops
Frost	4096x PPC-440 @ 700 MHz	8192	2 TB	22936
Blacklight	512x Xeon X7560 @ 2.3 GHz	4096	32 TB	36864
Trestles	1296x Opteron 6136 @ 2.4 GHz	10368	21 TB	100000

NCAR/University of Colorado

Machine	CPU	cores	RAM	Gflops
Janus	2736x Xeon X5660 @ 2.8 GHz	16416	32 TB	184000

An Implementation

Janus Currently 44 on Top500 list

- Power Supply: 2 MW ($\approx$ \$1,900 USD per day)
- Network: Fully non-blocking 40 Gbps QDR Infiniband
- Cooling: 81,000 gallons chilled water

Testing Communications: Asynchronous Ping Pong

- We test network latency and throughput between cores in various configurations across the network. (They play ping pong.)

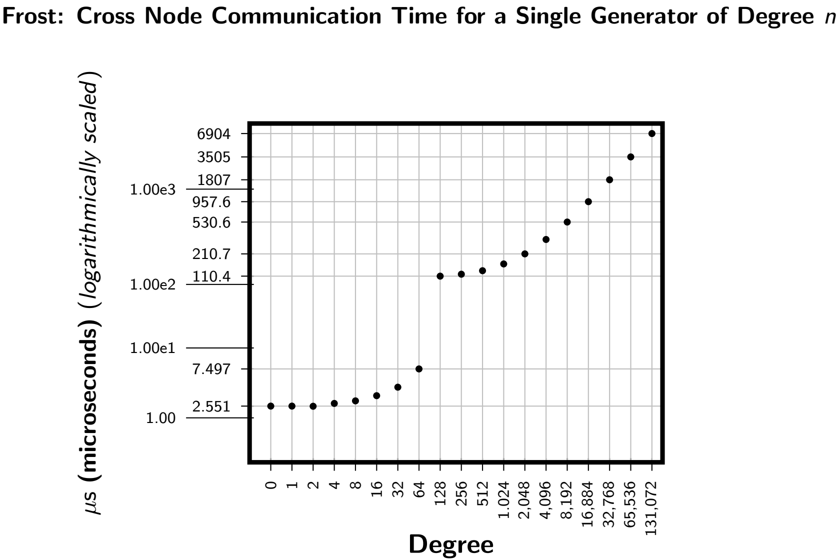
Testing Communications: Asynchronous Ping Pong

- We test network latency and throughput between cores in various configurations across the network. (They play ping pong.)
- This tells us how quickly we will be able to communicate generators for groups of varying degrees between cores.

	T_s (μs)	α (cycles)	throughput (MiB/s)	T_c (s/byte)	β (cycles/byte)
Frost (Single Node)	2.39	1675.80	1661.88	5.74×10^{-10}	4.02×10^{-1}
Frost (Cross Node)	2.55	1785.82	144.82	6.59×10^{-9}	4.61
Frost (Cross Partition)	2.84	1989.34	144.88	6.58×10^{-9}	4.61
Trestles (Single Node)	1.16	2775.62	968.55	9.85×10^{-10}	2.36
Trestles (Cross Node)	1.87	4491.48	2247.15	4.24×10^{-10}	1.02

Table 1: Single Byte Latency and Bandwidth by Machine and Communication Type

Testing Communications: Asynchronous Ping Pong



How to Proceed

Problem 1

- In general, we find that inter-processor communications are very expensive compared to intra-processor communications.
- The implication is that splitting a large backtrack search across thousands of cores on a supercomputer may take longer than splitting the same search on cores in a single processor.

How to Proceed

Problem 1

- In general, we find that inter-processor communications are very expensive compared to intra-processor communications.
- The implication is that splitting a large backtrack search across thousands of cores on a supercomputer may take longer than splitting the same search on cores in a single processor.

Problem 2

- Even if we do know how many cores to use, and which cores to assign specific tasks, we are still largely clueless as to how we should go about dividing the backtrack search itself.

How to Proceed

Problem 1

- In general, we find that inter-processor communications are very expensive compared to intra-processor communications.
- The implication is that splitting a large backtrack search across thousands of cores on a supercomputer may take longer than splitting the same search on cores in a single processor.

Problem 2

- Even if we do know how many cores to use, and which cores to assign specific tasks, we are still largely clueless as to how we should go about dividing the backtrack search itself.

The approach to solving these problems will vary by system architecture. It is best to consider an example.

An Example

Let $G = \langle a, b \rangle$ be the Fischer group $\mathrm{Fi}_{24}{}'$ of degree 306,936. This is the third largest sporadic group (behind M and B).

- $|a| = 2$
- $|b| = 3$
- $|G| = 2^{21} \cdot 3^{16} \cdot 5^{2} \cdot 7^{3} \cdot 11 \cdot 13 \cdot 17 \cdot 23 \cdot 29$

  $= 1{,}255{,}205{,}709{,}190{,}661{,}721{,}292{,}800$
- $g = ababbabbabb$ has order 6.
- We will find $C_G(g)$ on different platforms.
  - $|C_G(g)| = 559,872$
  - We will find 9 strong generators and a base of size 3.

Single Core

- On a single core machine, we do not have to worry about Problem 1 as there are no other cores to send messages to.
- We also don't need to worry about how we split the backtrack search.
- This is really just a slight modification of Leon's existing code.

Single Core

- On a single core machine, we do not have to worry about Problem 1 as there are no other cores to send messages to.
- We also don't need to worry about how we split the backtrack search.
- This is really just a slight modification of Leon's existing code.

```
./parcent 1 f24g1 fisch6 Cfisch6
```

Single Core

- On a single core machine, we do not have to worry about Problem 1 as there are no other cores to send messages to.
- We also don't need to worry about how we split the backtrack search.
- This is really just a slight modification of Leon's existing code.

```
./parcent 1 f24g1 fisch6 Cfisch6
```

BSGS construction time: +69.289662

SGS augmentation time: +122.772923

Single Core

- On a single core machine, we do not have to worry about Problem 1 as there are no other cores to send messages to.
- We also don't need to worry about how we split the backtrack search.
- This is really just a slight modification of Leon's existing code.

```
./parcent 1 f24g1 fisch6 Cfisch6
```

BSGS construction time: +69.289662

SGS augmentation time: +122.772923

`backtrack search time:` +25.211318

Dual Core

- Problem 1: We are able to stay within a single processor and communicate efficiently.
- Problem 2: We simply have the cores take every other node at some appropriate splitting level.

Dual Core

- Problem 1: We are able to stay within a single processor and communicate efficiently.
- Problem 2: We simply have the cores take every other node at some appropriate splitting level.

```
./parcent 2 f24g1 fisch6 Cfisch6
Attempting to use 2 cores
Using 1 parallel strategy on 2 cores
backtrack search time:  +14.269752
```

3-Core

- When we move to a 3-core calculation, we have some options.
- We may force a serial strategy:

3-Core

- When we move to a 3-core calculation, we have some options.
- We may force a serial strategy:

```
./parcent 3 -fs f24g1 fisch6 Cfisch6
Attempting to use 3 cores
Using 1 serial strategy on 1 core
Using 1 parallel strategy on 2 cores
Parallel strategy on 2 cores wins!
backtrack search time:  +14.336500
```

3-Core

- When we move to a 3-core calculation, we have some options.
- We may force a serial strategy:

```
./parcent 3 -fs f24g1 fisch6 Cfisch6
Attempting to use 3 cores
Using 1 serial strategy on 1 core
Using 1 parallel strategy on 2 cores
Parallel strategy on 2 cores wins!
backtrack search time:  +14.336500
```

- Or we may allow a 3-core parallel strategy:

```
./parcent 3 f24g1 fisch6 Cfisch6
Attempting to use 3 cores
Using 1 parallel strategy on 3 cores
backtrack search time:  +9.969995
```

8-Core on 2 CPUs

- For larger core counts spread across multiple CPUs, we limit communication between processors.
- We try a serial and a parallel strategy of the largest size possible.
- We assign the remaining cores on a single CPU with a randomized node strategy: A core considers orbits at splitting levels and assigns each orbit point randomly across the available cores.

8-Core on 2 CPUs

- For larger core counts spread across multiple CPUs, we limit communication between processors.
- We try a serial and a parallel strategy of the largest size possible.
- We assign the remaining cores on a single CPU with a randomized node strategy: A core considers orbits at splitting levels and assigns each orbit point randomly across the available cores.

```
./parcent 8 -fs f24g1 fisch6 Cfisch6
Attempting to use 8 cores
Using 1 serial strategy on 1 core
Using 1 parallel strategy on 4 cores
Using 1 random parallel strategy on 3 cores
Random parallel strategy on 3 cores wins!
backtrack search time: +5.709973
```

Massively Parallel Example

- We run on Janus with multiple random strategies across 4800 cores on 800 CPUs.
- Some strategies skim the top of the tree randomly.
- Others dive further into the tree before assigning a random splitting.

Massively Parallel Example

- We run on Janus with multiple random strategies across 4800 cores on 800 CPUs.
- Some strategies skim the top of the tree randomly.
- Others dive further into the tree before assigning a random splitting.

```
qsub parcent_fisch.pbs
Random parallel strategy on 12 cores wins!
backtrack search time:  +1.714144
```

This workshop hereby returns an exit status (modulo questions):

0