Managing the Masses

Mike Lewis
Senior Server Programmer, ArenaNet

AI ARTIFICIAL INTELLIGENCE SUMMIT

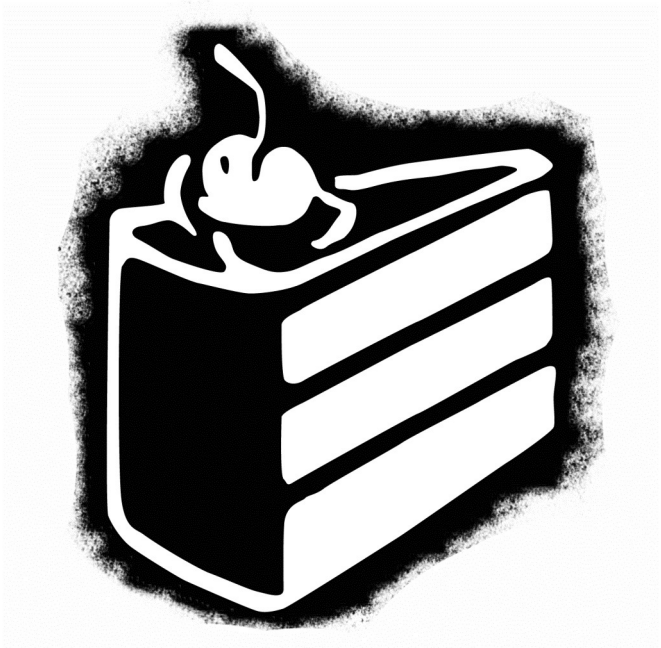
GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 5-9, 2012
EXPO DATES: MARCH 7-9
2012

GUILD WARS 2™

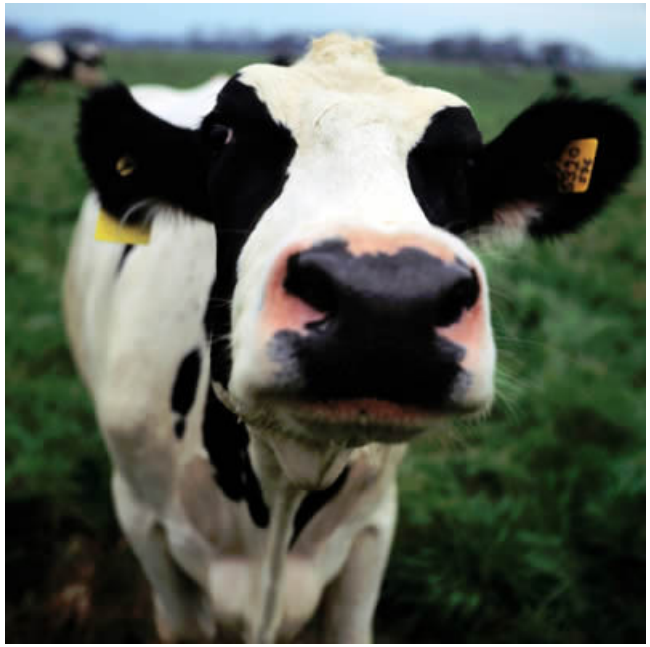
ARENANET®

Guild Wars 2

- Massively Multiplayer Online RPG
- Emphasis belongs on “Massive”
- Several hundred players in an environment
- Hundreds of AI agents per environment



Piece of Cake!



MMO Servers = Beefy!

How much beef, you ask?

A hypothetical server might look like...

- 12 cores at high clock speeds
- 32 GB of RAM
- Massive CPU caches
- Fast network, disk, etc.

Compare this to...

A commodity gaming PC/console

- 2-8 “heavy lifting” cores (plus SPUs, etc.)
- 1-2 dedicated GPU cores
- 512MB – 16GB of RAM
- Comparatively slow network and disks

Critical Comparison:
Resources Per Player

Resources Per Player (Xbox 360)

- 3 cores at ~3.2GHz
- Approximately 9.6GHz of raw processor
- Perhaps only one player per console
- All this goes to one person on the couch

So all this...

... goes to one (or maybe four) people.

Resources Per Player (MMOs)

- 12 cores at ~2.3GHz
- Approximately 27.6GHz of raw processor
- Conservatively, say 5000 players/server
- **Only 5.5MHz per player!**

For Perspective...

That’s roughly 5 Atari 2600s per person

OK, so it’s not quite that bad...

- More work per clock cycle on a modern CPU
- Overlapping data between players
- Shared computations throughout a map
- Some constant overheads
- Scaling per player isn’t necessarily linear

But we still have relatively limited resources per player.

Attacking the Problem: Guild Wars 2

Threading Architecture

- Asynchronous versus Synchronous layers
  - Async is the messy “stuff that can fail”
  - Sync is everything else, i.e. the game logic
- Game Contexts
  - Major public maps
  - Personal story instances

Threading Architecture

Game Contexts are “synchronous”

- A server runs many contexts in parallel
- But each context is effectively single threaded
- Remember resources per player

What runs on a core?

Asynchronous layer

- Network logic
- Database communication
- Security and sanity validations
- Sync layer coordination
- Spread across all cores

What runs on a core?

Synchronous layer

- Simulation/event framework
- Game rules and logic
- Physics
- Latency compensation
- Only one thread per context

This is starting to look pretty good...

AI Architecture

Highly optimized behavior trees

- Prioritized lists of actions to take
- Each “thought tick”, agents select one action
- Resemble telephone poles more than trees

Some customizable scripting capabilities also

Why this approach?

- Minimizes latency for game simulation
- Programming game code is simpler
- Increased security and stability
- The alternatives are all worse

Some Alternatives

- Vertical scaling
- Full multithreading
- Client-side offloading
- Distributed computation

Vertical Scaling

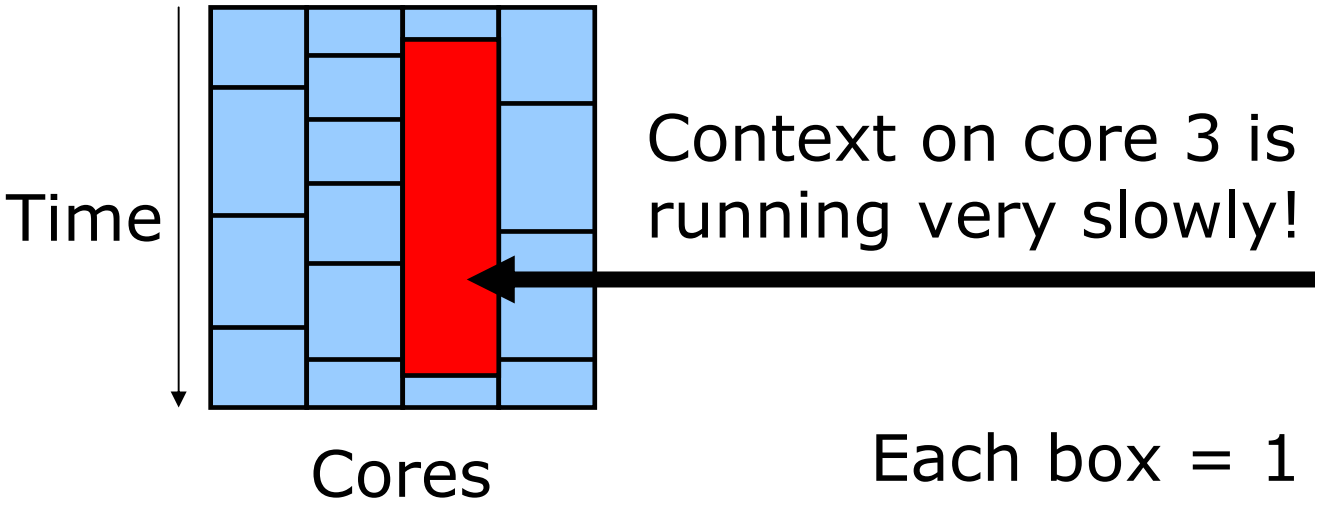
- Just add some more beef!
- Beef isn’t cheap
- Cost grows super-linearly
- Becomes prohibitively expensive very quickly

All-out Multithreading

- Don’t silo contexts onto single threads
- Use many threads per context instead
- Increases total latency
- Much more difficult to write robust code

All-out Multithreading

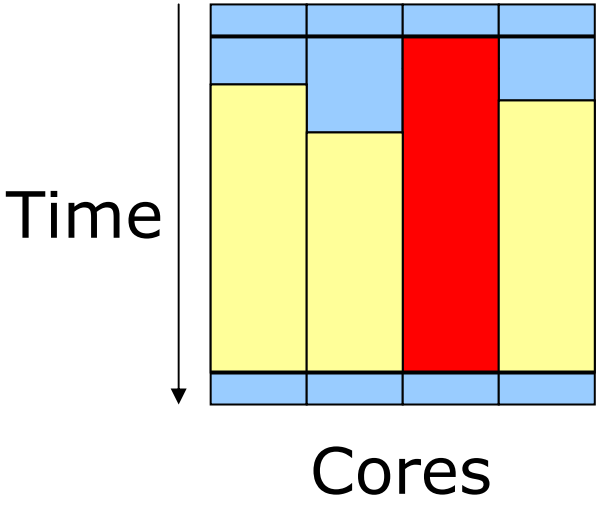
Latency with a core per context:



Each box = 1 game tick

All-out Multithreading

Latency with multiple threads per context:



Thread 3 stalls, leaving all other cores idle until context is completely done!

Each box = 1 game tick

All-out Multithreading

- This can be amortized with thread pooling
- But average performance is often worse
- Much harder to code for robustly
- Hard to coordinate with async layer as well

Client-side Offloading

- Leave some computations for the client
- Lots of clients connected = lots of CPUs
- Very risky from security perspective
- How to handle disconnected players?

Practical Client-side Offloading

- Leave “non-risky” stuff on the client
- Perform security-sensitive stuff on servers
- Standard approach for multiplayer games
- Has definite benefits but only goes so far

Distributed Server Model

- Split server-side load between machines
- One “game context” is now many servers
- Allows easy horizontal scaling
- Need more CPU time? Add another server!

Distributed Server Model

- Not using shared-state concurrency
- Safer and more reliable in many ways
- But also difficult in its own right
- Lots of things are essentially impossible

Distributed Server Model

Hundreds of Impossibility Results for Distributed Computing

Faith Fich and Eric Ruppert

February 26, 2003

http://bit.ly/kLutYX

Managing the Masses

- MMOs are big. Really big...
- Lots of server power, but little per player
- Threading model makes all the difference
- Lots of room for future innovation

Thank You!