Autotuning Aspects of PetaBricks

Jason Ansel

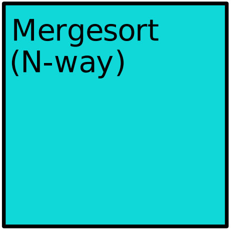
Collaboration with:
Una-May O'Reilly   Alan Edelman   Saman Amarasinghe

MIT - CSAIL

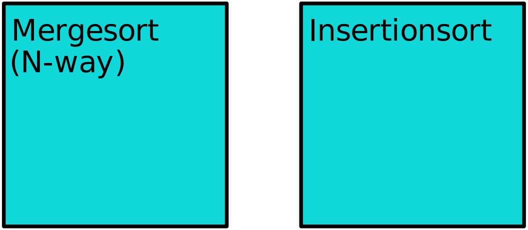
April 3, 2011

Outline

Algorithmic choice



Algorithmic choice



Algorithmic choice

Mergesort (N-way)

Insertionsort

Radixsort

Algorithmic choice

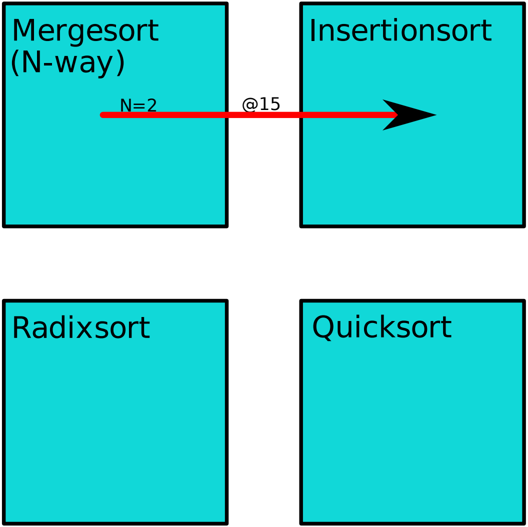
Mergesort (N-way)

Insertionsort

Radixsort

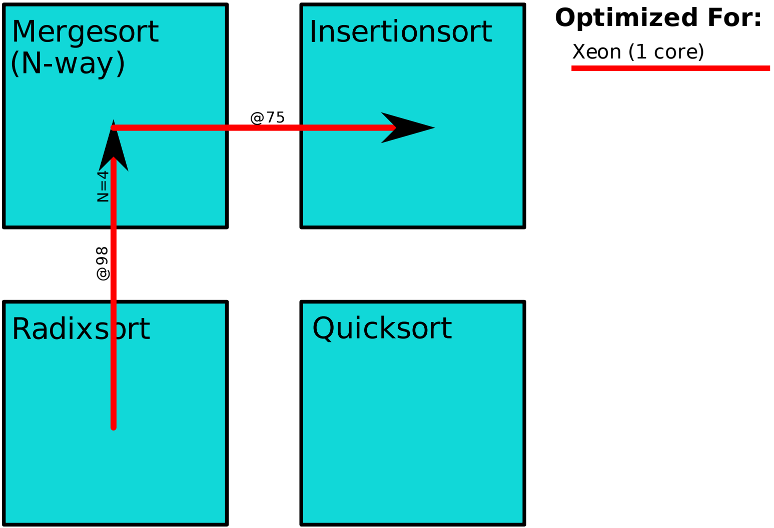
Quicksort

Algorithmic choice

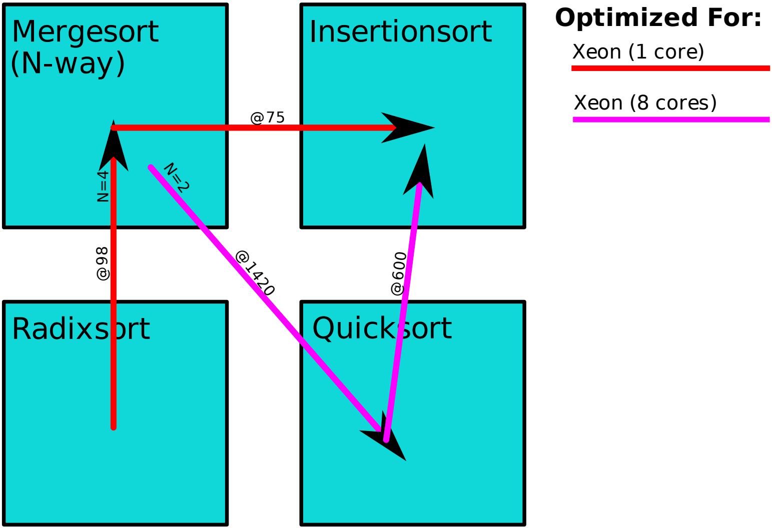


STL Algorithm

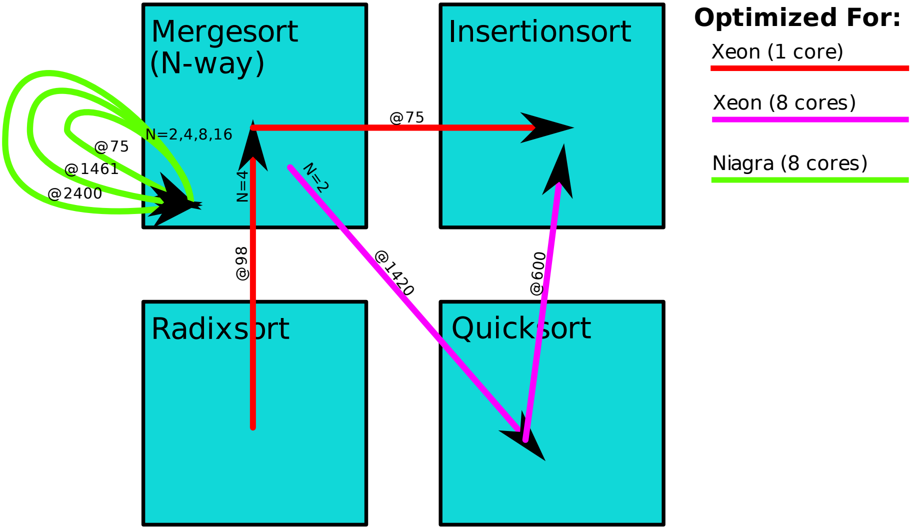
Algorithmic choice



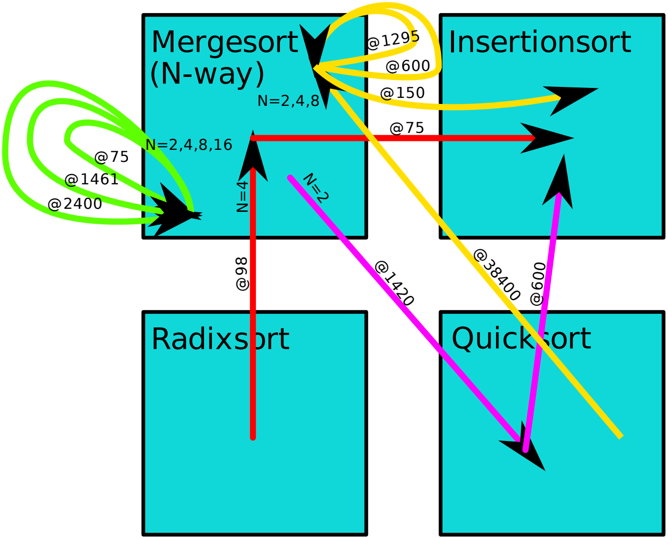
Algorithmic choice



Algorithmic choice



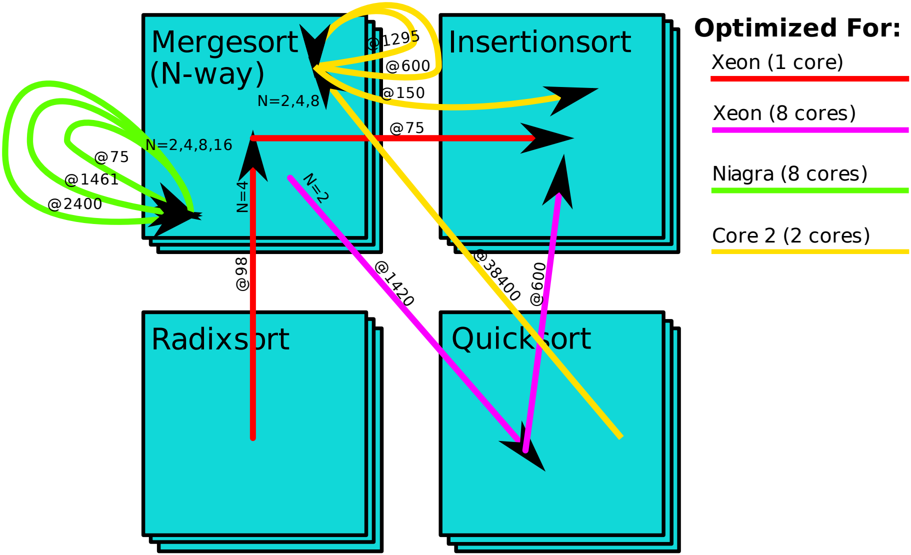
Algorithmic choice



Optimized For:

- Xeon (1 core)
- Xeon (8 cores)
- Niagra (8 cores)
- Core 2 (2 cores)

Algorithmic choice



Outline

Algorithmic choices

Language

```
either {
  InsertionSort(out, in);
} or {
  QuickSort(out, in);
} or {
  MergeSort(out, in);
} or {
  RadixSort(out, in);
}
```

Algorithmic choices

Language

```
either {
  InsertionSort(out, in);
} or {
  QuickSort(out, in);
} or {
  MergeSort(out, in);
} or {
  RadixSort(out, in);
}
```

$\Rightarrow$

Representation

Decision tree synthesized by our evolutionary algorithm (EA)

The PetaBricks language

- Choices expressed in the language
  - High level algorithmic choices
  - Dependency-based synthesized outer control flow
  - Parallelization strategy
- Programs automatically adapt to their environment
  - Tuned using our bottom-up evaluation algorithm
  - Offline autotuner or always-on online autotuner

Variable accuracy (quality of service) choices

Language

accuracy_metric MyRMSError

Variable accuracy (quality of service) choices

Language

```
accuracy_metric MyRMSError
...
for_enough {
  SORIteration(tmp);
}
```

Variable accuracy (quality of service) choices

Language

```
accuracy_metric MyRMSError
...
for_enough {
  SORIteration(tmp);
}
```

$\Rightarrow$

Representation

Function from problem size to number of iterations synthesized by our EA

Variable accuracy (quality of service) choices

Language

```
accuracy_metric MyRMSError
...
for_enough {
  SORIteration(tmp);
}
```

⇒

Representation

Function from problem size to number of iterations synthesized by our EA

- For more see our paper in CGO'11

Large choice space

Benchmark	Variable accuracy	Search space dimensions
Bin Packing	Yes	117
Clustering	Yes	91
Eigenproblem	No	35
Helmholtz	Yes	61
Image Compression	Yes	163
LU Factorization	No	140
Matrix Multiply	No	108
Poisson	Yes	64
Preconditioner	Yes	159
Sort	No	33
Average	-	97.1

Outline

Traditional evolution algorithm

Initial population	?	?	?	?	Cost $= 0$

Traditional evolution algorithm

Initial population	72.7s	?	?	?	Cost = 72.7

Traditional evolution algorithm

Initial population	72.7s	10.5s	?	?	Cost $= 83.2$

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	?	Cost = 87.3

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost $= 118.5$

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	?	?	?	?	Cost = 0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost $= 118.5$
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost $= 25.1$

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	?	?	?	?	Cost = 0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	?	?	?	?	Cost = 0

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	0.3s	0.1s	0.4s	2.4s	Cost = 3.2

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost = 118.5
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost = 25.1
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost = 9.0
Generation 4	0.3s	0.1s	0.4s	2.4s	Cost = 3.2

- Cost of autotuning front-loaded in initial (unfit) population
- We could speed up tuning if we start with a faster initial population

Traditional evolution algorithm

Initial population	72.7s	10.5s	4.1s	31.2s	Cost $= 118.5$
Generation 2	4.2s	5.1s	2.6s	13.2s	Cost $= 25.1$
Generation 3	2.8s	0.1s	3.8s	2.3s	Cost $= 9.0$
Generation 4	0.3s	0.1s	0.4s	2.4s	Cost $= 3.2$

- Cost of autotuning front-loaded in initial (unfit) population
- We could speed up tuning if we start with a faster initial population

Key insight

Smaller input sizes can be used to form better initial population

Bottom-up evolutionary algorithm

- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 1, to form initial population for:
- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

Bottom-up evolutionary algorithm

- Train on input size 1, to form initial population for:
- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

- Naturally exploits optimal substructure of problems

Bottom-up evolutionary algorithm

- Train on input size 1, to form initial population for:
- Train on input size 2, to form initial population for:
- Train on input size 8, to form initial population for:
- Train on input size 16, to form initial population for:
- Train on input size 32, to form initial population for:
- Train on input size 64

- Naturally exploits optimal substructure of problems

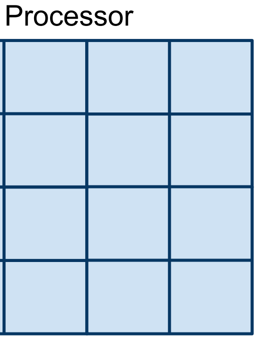
- For more see our paper to appear in GECCO'11

Outline

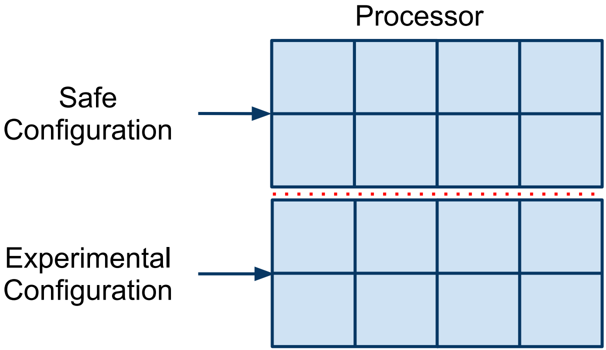
Challenges for online learning

- Search space is difficult to model
  - High-dimensional
  - Non-linear
  - Irregular
  - Complex dependencies
- Dangerous configurations exist
  - Exponential algorithms
  - Infinite loops
  - Poor quality of service

SiblingRivalry (online autotuner)



SiblingRivalry (online autotuner)



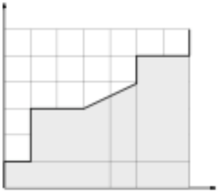
SiblingRivalry (online autotuner)

- Split available resources in half
- Process identical requests on both halfs
- Race two candidate configurations (safe and experimental) and terminate slower algorithm
- Initial slowdown (from duplicating the request) can be overcome by autotuner
- Surprisingly, reduces average power consumption per request

Learning technique

- Maintain population of candidate algorithms
- Each candidate must be pareto-optimal in 3D objective space:
  - Performance
  - Quality of service
  - Confidence
- Pick safe and experimental configurations from population
- Mutate the experimental configuration
- Add the new configuration to the population if it wins the race

Adaptive operator selection

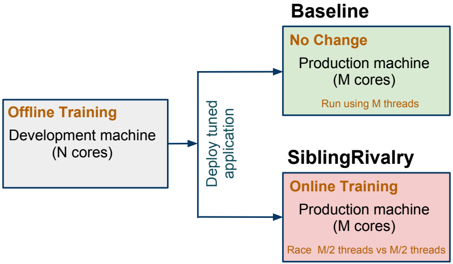


- Extension of bandit-based differential evolution [DaCosta et al.]
- Deterministically choses mutation operators
- Requires only relative performance information
- Considers trade-off between *exploitation* and *exploration*

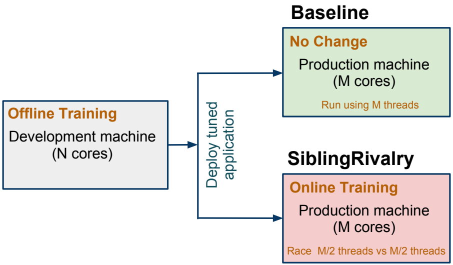
$$\arg\max_i \left(AUC_i + C\sqrt{\frac{2\log\sum_k n_k}{n_i}} \right)$$

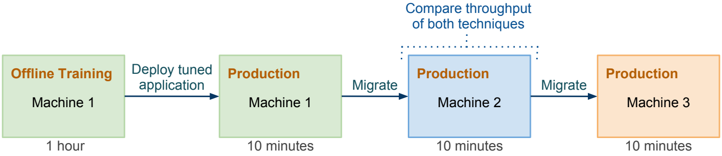
Outline

Experimental setup

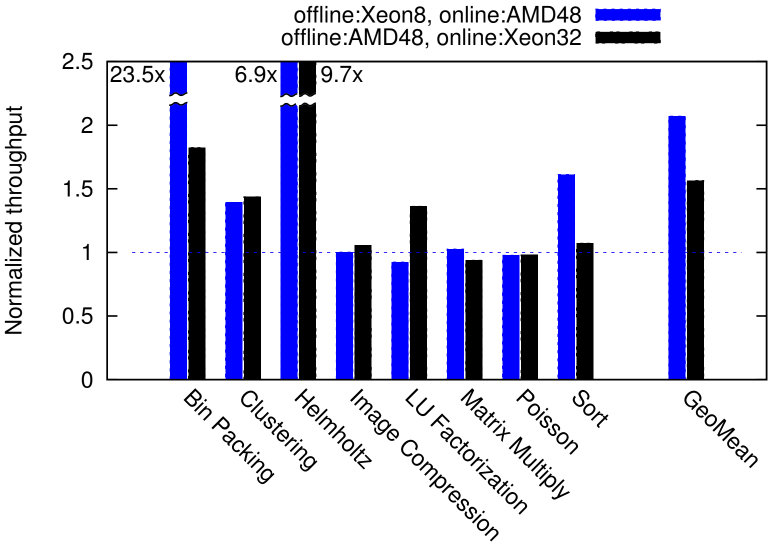


Experimental setup

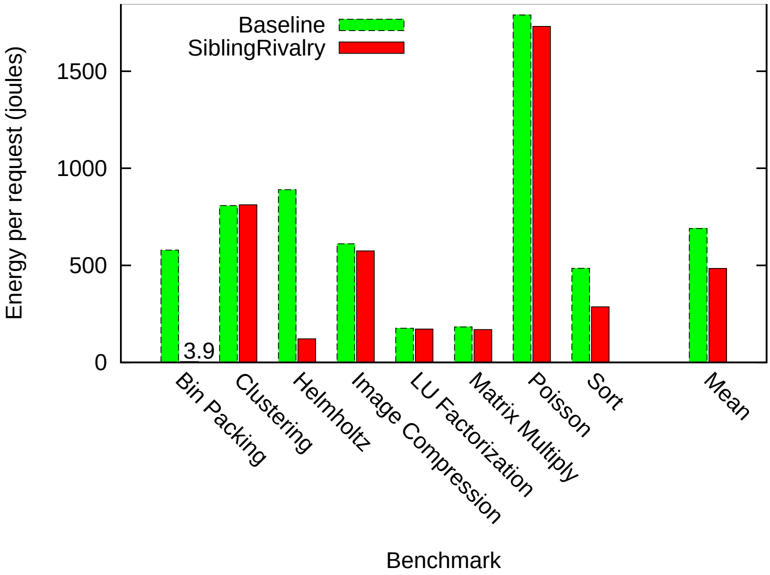




SiblingRivalry: throughput



SiblingRivalry: energy usage (on AMD48)



Outline

Conclusions

Motivating goal of PetaBricks

Make programs future-proof by allowing them adapt to their environment.

Thanks!

- Questions?
- `http://projects.csail.mit.edu/petabricks/`

Backup slides

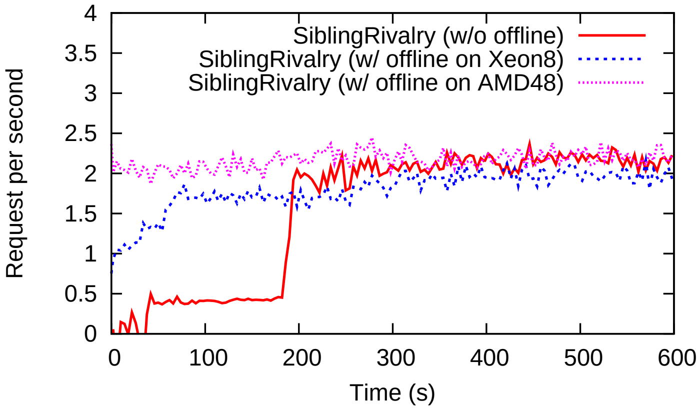
Current and future work

- Publications
  - PetaBricks: A Language and Compiler for Algorithmic Choice. [PLDI'09]
  - Autotuning Multigrid with PetaBricks. [SC'09]
  - Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms. [CGO'11]
  - An Efficient Evolutionary Algorithm for Solving Bottom Up Problems [GECCO'11]
- Under review
  - SiblingRivalry: Online Autotuning Through Local Competitions
- Current projects
  - Improving bandit-based operator selection
  - Modeling our search space
  - Heterogeneous systems

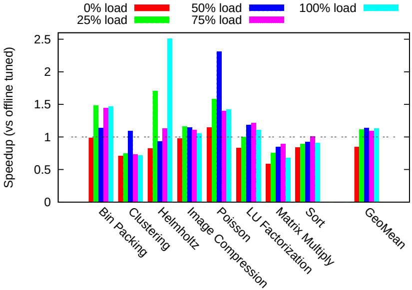
Test systems

Acronym	Processor Type	Processors
Mobile	Core 2 Duo Mobile 1.6 GHz	1 ($\times$2 cores)
Niagara	Sun Fire T200 Niagara 1.2 GHz	1 ($\times$8 cores)
Xeon1	Intel Xeon X5460 3.16GHz	1 (other cores disabled)
Xeon8	Intel Xeon X5460 3.16GHz	2 ($\times$4 cores)
Xeon32	Intel Xeon X7560 2.27GHz	4 ($\times$8 cores)
AMD48	AMD Opteron 6168 1.9GHz	4 ($\times$12 cores)

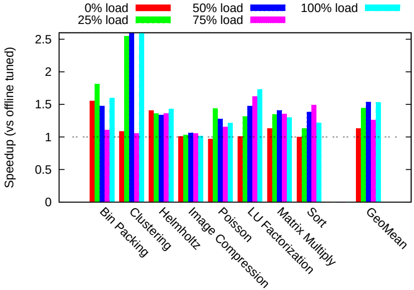
SiblingRivalry: convergence (Sort on AMD48)



SiblingRivalry: adapting to load



Xeon8



AMD48