Estimating Parallel Compute Performance for Eilmer Simulations

Rowan Gollan

03 October 2019

Outline

- A model of parallel execution
- Parallel execution in Eilmer
- Strategies for load-balancing
- An approach to estimate parallel performance
- Some examples:
  - double cone
  - flared cone
  - JCEAP configuration

A model of performance for parallel execution

- How does execution time and efficiency vary with increasing processor count?
- *What is the fastest I can solve my problem on a cluster with many cores available?*
- fixed-size problem analysis, or strong scaling

Execution time of an algorithm on a single processor:

$$T_1 = sT_1 + pT_1$$

where:

- s: serial part of the algorithm; *no benefit* from increasing processor count
- p: parallel part of the algorithm; work divides perfectly across more processors

Speed-up with multi-processor computer

Execution time of an algorithm on a single processor:

$$T_1 = sT_1 + pT_1$$

Execution time of same algorithm on n processors:

$$T_n = sT_1 + \frac{p}{n}T_1$$

Speed-up is how much faster the algorithm executes on n processors compared to a single processor:

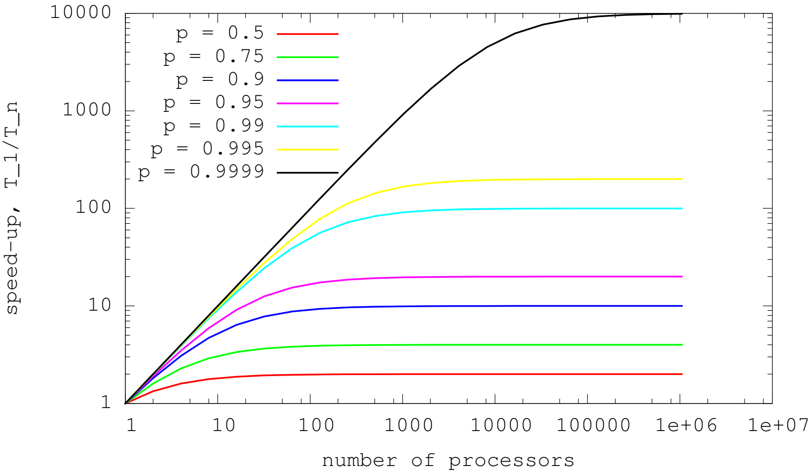
$$S = \frac{T_1}{T_n}$$

For large n, speed-up is limited by serial fraction of algorithm to:

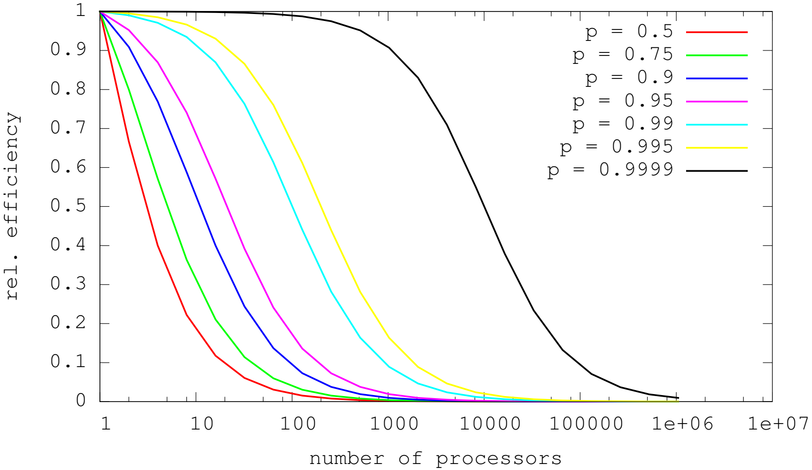
$$S = \frac{1}{s} = \frac{1}{1-p}$$

The consequence is we require a very high fraction of parallel work to get benefit from increasing the number of processors we use.

Limits on Speed-up: Amdahl's Law



Limits on Efficiency



$$E_{rel} = \frac{T_1}{nT_n}$$

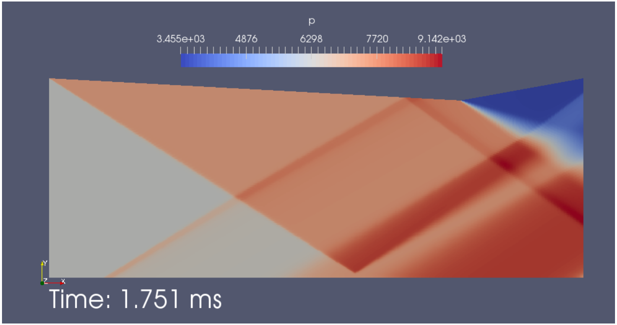
Parallel Execution in Eilmer

We use domain decomposition to increase the parallel fraction of the algorithm in Eilmer.

$$T_n = sT_1 + \frac{p}{n}T_1$$

s: global coordination activities; global decisions; I/O; communication; waiting on other processors to finish

p: timestep update on a block or collection of blocks

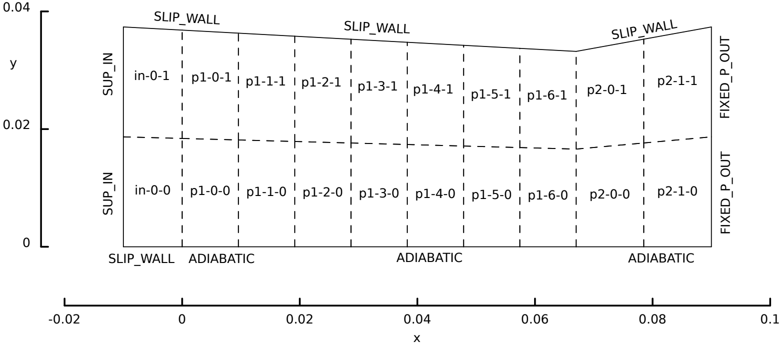


Parallel Execution in Eilmer

We use domain decomposition to increase the parallel fraction of the algorithm in Eilmer.

$$T_n = sT_1 + \frac{p}{n} T_1$$

- *s*: global coordination activities; global decisions; I/O; communication; waiting on other processors to finish
- *p*: timestep update on a block or collection of blocks



Load-balancing options in Eilmer

Balancing the compute work across the processors is key to maintaining a high parallel fraction. If processors finish work early, they are idle. Idle time contributes to serial fraction.

Options for load balancing in Eilmer:

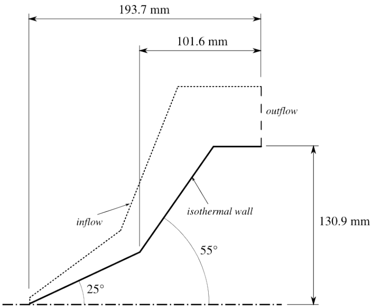
- manual: build by hand blocks of roughly equal workload (ie. number of cells)
- `FBArray`: the fluid-block-array object can carve up a large (hand-built) block into many smaller blocks of equal work
- load-balance tool: assign collections of blocks to processors, distribute the collections of blocks to try to equalise work on the processors
- `Metis`: 3rd-party tool to partition unstructured grids

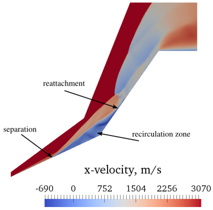
A suggested approach for estimating parallel fraction

- parallel fraction varies due to: number of cells, how well load balanced, patterns of block-to-block communication
- easiest to determine parallel fraction by small time trials

1. Time the execution for 1000 steps (for example) on n_1 processors. Record the time spent on time-stepping. Ignore the start-up cost.
2. Time the execution for 1000 steps on n_2 (for example $n_2 = 2n_1$). Record the time spent on time-stepping.
3. Compute speed-up from using n_1 to using n_2:
$$S = \frac{T_{n1}}{T_{n2}}$$
4. Estimate the parallel fraction as:
$$p = \frac{S - 1}{S - 1 + \frac{1}{n_1} - \frac{S}{n_2}}$$
5. Repeat for $n_3, n_4, \ldots$ until you are satisfied you have a clear picture of the parallel fraction

Double cone






```
FBArray:new{nib=..., njb=...}
```

Double cone

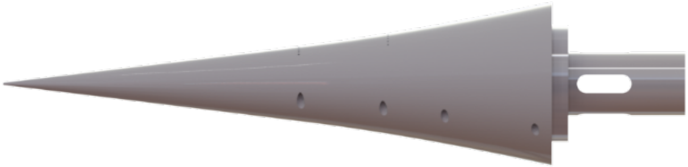
number of cells: 192 k

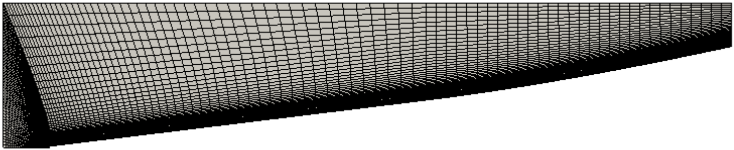
load-balance: 1 block per core, domain partitioned with `FBArray` objects

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
70	89.0	-	-	-
140	43.4	2.05	1.00034	1.02534
280	22.7	1.9119	0.9997	0.9559
560	10.5	2.1619	1.00024	1.0809
1120	7.0	1.5	0.9991	0.75
2240	5.25	1.333	-	0.6667

Flared cone

Thanks to Isaac Convery-Brien for test case and images.





```
> e4loadbalance --job=flared-cone --ntasks=28
```

Flared cone

number of cells: ≈ 140 k

load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.2	1.9934	-	0.9967

Flared cone

number of cells: $\approx$ 140 k

load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.2	1.9934	-	0.9967
112	15.9	1.89937	0.99900	0.94968

Flared cone

number of cells: $\approx$ 140 k

load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.9	1.89937	0.99900	0.94968
224	8.8	1.8171	-	0.9086

Flared cone

number of cells: $\approx$ 140 k

load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.9	1.89937	0.99900	0.94968
224	8.8	1.8171	-	0.9086
224	8.0	1.9875	0.99994	0.99375

Flared cone

number of cells: $\approx$ 140 k

load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.9	1.89937	0.99900	0.94968
224	8.0	1.9875	0.99994	0.99375
448	4.1	1.9754	-	0.9877

Flared cone

number of cells: ≈ 140 k

load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.9	1.89937	0.99900	0.94968
224	8.0	1.9875	0.99994	0.99375
448	4.1	1.9754	-	0.9877
448	4.5	1.7777	0.99936	0.8888

Flared cone

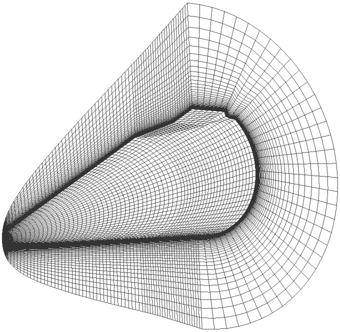
number of cells: $\approx$ 140 k

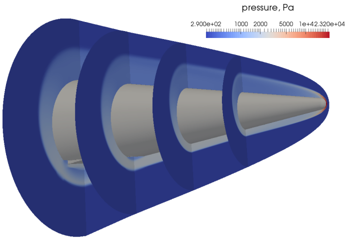
load-balance: 448 blocks for every case, distributed with load-balance tool

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	60.3	-	-	-
56	30.2	1.9967	0.99994	0.99834
112	15.9	1.89937	0.99900	0.94968
224	8.0	1.9875	0.99994	0.99375
448	4.5	1.7777	0.99936	0.8888
896	2.75	1.6363	-	0.8181

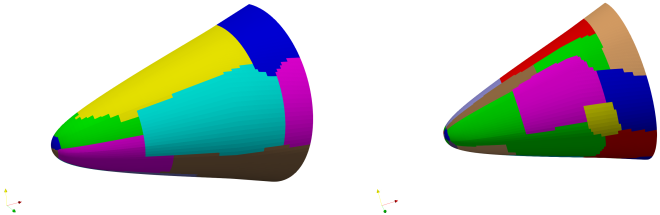
JCEAP configuration

Thanks to Kyle Damm for test case and images.





JCEAP configuration



```
> ugrid_partition jceap.su2 mapped_cells 28 3
```

JCEAP configuration

number of cells: $\approx$ 700 k

load-balance: 1 block per core, domain partitioned with `Metis`

no. cores	time (s) for 1000 steps	speed-up	p	rel. efficiency
28	272.1	-	-	-
56	138.3	1.9674	0.9994	0.9837
112	74.5	1.8563	0.9985	0.9282
224	38.9	1.9152	0.9996	0.95758
448	23.5	1.6553	0.9988	0.8277
896	15.8	1.4873	-	0.7437

Concluding remarks

- Load balance is critical for good parallel performance
- Eilmer provides several methods to make load balancing easy:
  - `FBArray`: structured grids, relatively simple geometries
  - `e4loadbalance`: when blocking (topology) is dictated by geometry
  - `ugrid_partition`: using Metis for unstructured grids
- Some simple time trials can give you a quick estimate on how your simulation will scale given your problem size and the cluster you are using
- Eilmer shows very good strong scaling on modern clusters when care is taken with load balance