Final HPC Assigment

Università degli Studi di Trieste

27th february 2024

Exercise 1

The goal of the exercise was to estimate the latency of two OpenMPI collective operation, one of which being *Broadcast*. My work focussed on studying

- Broadcast
- Barrier

All of these algorithms were studied varying between four implementations, and varying process allocation between *core*, *socket* and *node*.

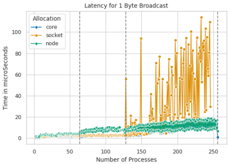
Broadcast

I've studied four different implementation of the algorithm:

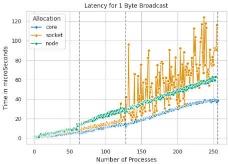
- Default
- Basic Linear
- Chain
- Binary tree

Having many degrees of freedom to analyze, I've decided to sequentially fix some of them, in order to see the impact of few on the *Avg Latency* variable.

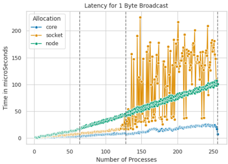
Fixing message size of 1 Byte



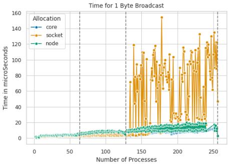
(a) Average Latency vs n. processes in default algorithm



(b) Average Latency vs n. processes in linear algorithm

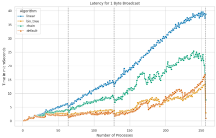


(c) Average Latency vs n. processes in chain algorithm

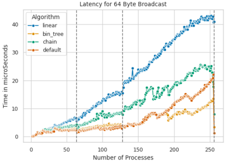


(d) Average Latency vs n. processes in binary tree algorithm

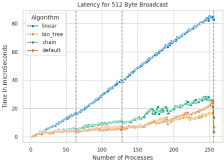
Fixing *core* allocation and varying message size



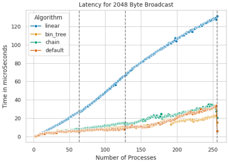
(a) Latency vs n. processes by algorithm



(b) Average Latency vs n. processes for 64 byte message



(c) Average Latency vs n. processes in 512 byte message



(d) Average Latency vs n. processes in 2024 byte message

Model

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-68.0031	5.0067	-13.58	0.0000
Algorithmchain	48.3330	4.5653	10.59	0.0000
Algorithmdefault	4.7424	4.5774	1.04	0.3002
Algorithmlinear	12.0579	4.6361	2.60	0.0093
Allocationnode	39.0153	3.9529	9.87	0.0000
Allocationsocket	63.3617	3.9943	15.86	0.0000
Processes	0.5054	0.0225	22.49	0.0000
MessageSize	0.0013	0.0027	0.46	0.6452

Model analysis with all variable included

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-11.7242	0.2489	-47.10	0.0000
Algorithmchain	5.3313	0.2587	20.60	0.0000
Algorithmdefault	0.5461	0.2587	2.11	0.0348
Algorithmlinear	23.9312	0.2587	92.49	0.0000
Processes	0.1198	0.0012	96.41	0.0000
MessageSize	0.0080	0.0002	51.66	0.0000

Model analysis with fixed allocation

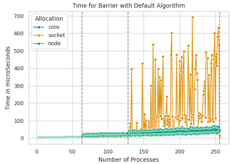
Barrier

Also here I've analyzed four different implementations:

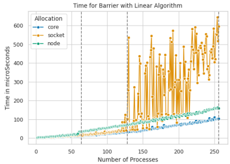
- Default
- Linear
- Double Ring
- Bruck algorithm

The same analysis steps of the previous algorithm apply

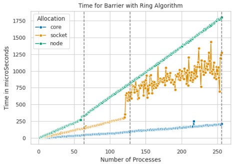
Vary allocation



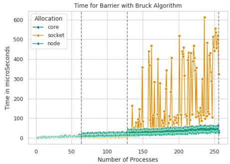
(a) Average Latency vs n. processes in default algorithm



(b) Average Latency vs n. processes in linear algorithm



(c) Average Latency vs n. processes in double ring algorithm



(d) Average Latency vs n. processes in bruck algorithm

Fixing Allocation

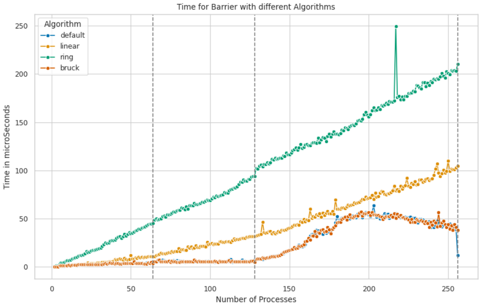


Figure: Latency vs n. processes by algorithm

Performance Model

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-293.2244	11.7596	-24.93	0.0000
Algorithmdefault	2.5873	11.0436	0.23	0.8148
Algorithmlinear	45.7123	11.0436	4.14	0.0000
Algorithmring	449.1283	11.0436	40.67	0.0000
Allocationnode	196.4844	9.5640	20.54	0.0000
Allocationsocket	178.5540	9.5640	18.67	0.0000
Processes	1.6650	0.0530	31.39	0.0000

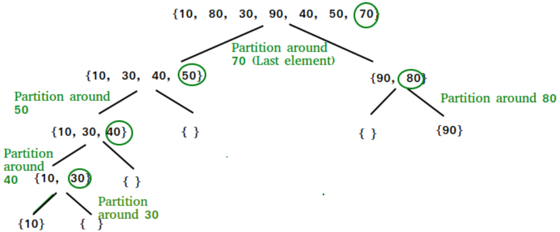
Model with all parameters

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-34.3959	1.5941	-21.58	0.0000
Algorithmdefault	0.0435	1.6956	0.03	0.9795
Algorithmlinear	19.0073	1.6956	11.21	0.0000
Algorithmring	78.4787	1.6956	46.28	0.0000
Processes	0.4335	0.0081	53.23	0.0000

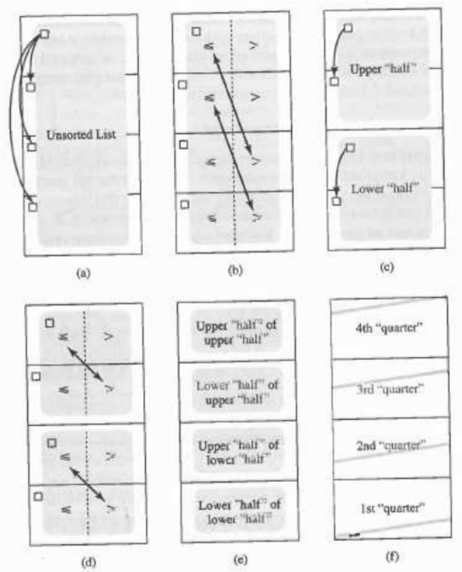
Model with fixed allocation

Exercise 2b

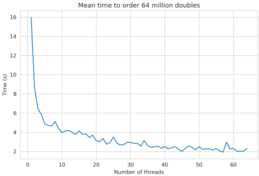
The aim of the exercise was to implement a parallel version of the Quicksort routine through the use of both OpenMP and MPI, analyzing its Scalability and performance.

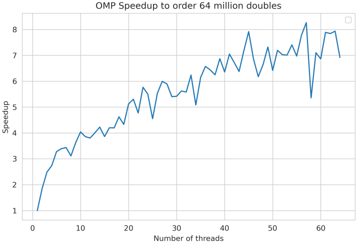

{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

Figure

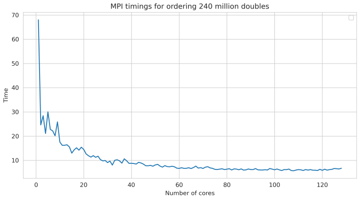

Unsorted List
(a)
≤
>
(b)
Upper "half"
Lower "half"
(c)
(d)
Upper "half" of upper "half"
Lower "half" of upper "half"
Upper "half" of lower "half"
Lower "half" of lower "half"
(e)
4th "quarter"
3rd "quarter"
2nd "quarter"
1st "quarter"
(f)

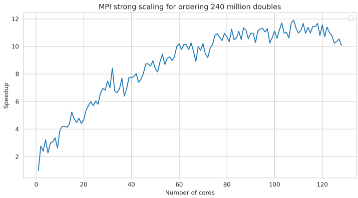
OpenMP scalability





Strong MPI scalability





Weak MPI Scalability

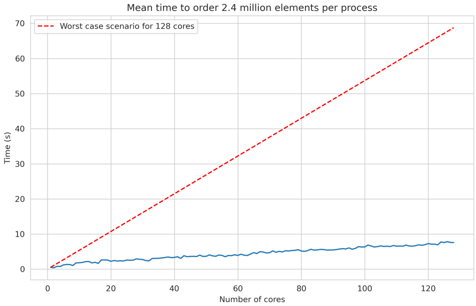


Figure: Time vs #cores at constant workload

Conclusions and Improvement

Even though the algorithm doesn't scale linearly, it does what expected, and provides decent results, easing computational time of a very efficient sorting routine as Quicksort.
Possible improvements:

- Try and see if a completely random pivot would work better than selecting a "good" one.
- Try and implement an algorithm with a linear number of communications with respect to number of processes
- Implement a similar approach also in shared memory
- Try a different partitioning routine which could exploit parallelism better