Paradigmas de Computação Paralela

Optimising performance (MPI)

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

10 Nov 2015

Performance of parallel applications

Performance models

- **Make it possible to compare algorithms, scalability and the identification of bottlenecks before a considerable time is invested in implementation**

What is the definition of performance?

- There are multiple alternatives:
  - Execution time, efficiency, scalability, memory requirements, throughput, latency, project costs / development costs, portability, reuse potential
  - The importance of each one depends on the concrete application
- Most common measure in parallel applications (*speed-up*): tseq/tpar

Amdahl law

- The sequential component of an application limits the maximum speed-up
  - If *s* is the sequential faction of an algorithm then the maximum possible gain is *1/s*.
- Reinforces the idea that we should prefer algorithms suitable for parallel execution: *think parallel*.

Performance of parallel applications

Performance models

- Should explain observations and predict behaviour
  - Defined as a function of the problem dimension, number of processors, number of tasks, etc.
- ***Execution time***
  - Time measured since the first processor (core) starts execution until the last processor terminates
  - **T*exec* = T*comp* + T*comm* + T*free***
  - ***Computation time* –** time spent in computations, excluding communication/synchronization and free time.
    - The sequential version can be used to estimate T*comp*.
  - ***Free time*** - when a processor becomes starved (without work)
    - Can be complex to measure since it depends on the order of tasks
    - Can be minimized with adequate load distribution and/or "sob-positioning" computation and communication

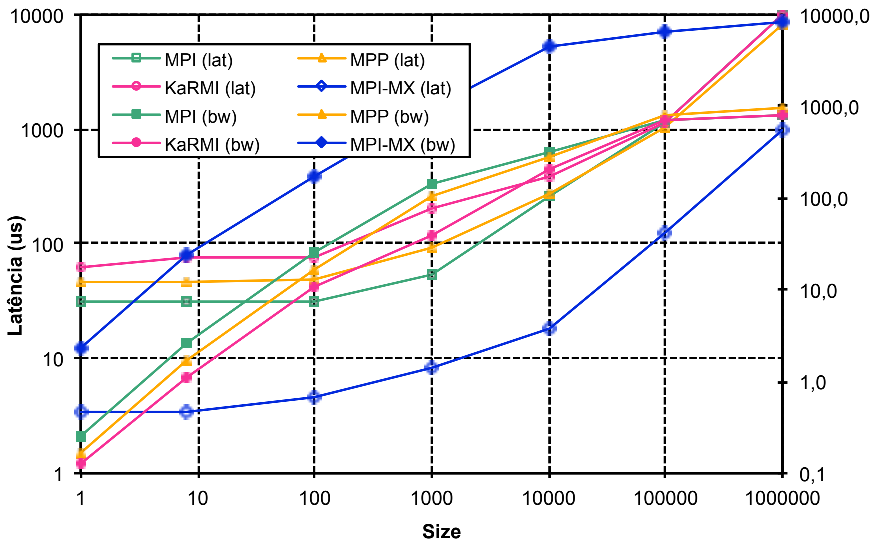
Performance of parallel applications

Performance models(cont.)

- ***Communication time*** – time that processes spend sending/receiving data.
  - Computed using communication latency (t*s*) and throughput (1/t*w*):
    - **T*mens* = ts + tw*L***

t*s* and t*w* can be obtained experimentally, by a ping-pong test and a linear regression.

Tamnho	MPI (lat)	MPI (bw)	MPI-MX (lat)	MPI-MX (bw)	MPP (lat)	MPP (bw)	KaRMI (lat)	KaRMI (bw)
1	31	0,3	3,4	2,3	46	0,2	63	0,1
8	31	2,6	3,4	23,7	46	1,7	75	1,1
100	31	25,6	4,7	168,7	48	16,6	75	10,7
1000	55	146,3	8,1	983,9	94	106,4	200	40,0
10000	258	310,3	18,4	4355,9	279	286,0	387	206,0
100000	1136	703,8	125,5	6373,0	1017	786,5	1137	703,0
1E+06	9859	811,4	970,2	8246,0	8282	953,2	9787	817,0
ts (us)	31		3		46		63	
tw (us)	0,010		0,001		0,008		0,010	

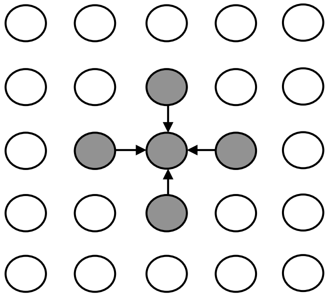


Performance of parallel applications

Performance models – Example Jacobi Method

- Iterative method, at each iteration the new matrix value is computed as the average neighbour values

$$X_{i,j}^{(t+1)} = aX_{i,j}^{(t)} + b(X_{i-1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j+1}^{(t)})$$



```
for(int t=0; t<Niter; t++) {
    for(int i=1; i<N-1; i++)
        for(int j=1; j<N-1; j++)
            r[i][j] = a*x[i][j]+ b*(x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]);
    // x = r on the next iteration
}
```

Performance of parallel applications

Performance models – Example Jacobi Method

- Execution time for each iteration in Jacobi method, for a NxN matrix, on P processors, a partition by columns and N/P columns per processor.

  T*comp* = operations per element *x* num. elements per processor *x* tc

  = 6 *x* (N *x* N/P) *x* t*c* (t*c* = time for a single operation)

  = 6t*c*N^2/P

  T*comm* = messages per processor *x* time required for each message

  = 2 *x* (t*s* + t*w*N)

  T*free* = 0 , since in this problem the workload is well distributed

T*exec* = T*comp* + T*comm* + T*free*

= 6t*c*N^2/P + 2t*s* + 2t*w*N

= O(N^2/P+N)

Performance of parallel applications

Performance models – Example (cont)

- In certain cases, execution time may not be the most adequate performance measure.
- Speed-up and efficiency are two related metrics.
- **Speed-up (G)** indicates the reduction in execution time attained in P processors
  - Ratio between the *best sequential algorithm* and the execution time of the parallel version

speed-up = T*seq* / T*par*,

- ***Efficiency*** **(E)** gives the faction of time that processors perform useful work:

E = T*seq* / (P *x* T*par*)

- Jacobi case:

$$G = \frac{6t_c N^2 P}{6t_c N^2 + 2Pt_s + 2Pt_w N} \qquad E = \frac{6t_c N^2}{6t_c N^2 + 2Pt_s + 2Pt_w N}$$

Performance of parallel applications

Scalability analysis

- Execution time, speed-up and efficiency can be used for quantitative analysis of performance
- Jacobi example:
  - **Execution time decreases when P increases, but it is limited by the time to exchange two lines**
  - **Execution time increases with N, t*c*, t*s* e t*w***
  - **Efficiency decreases when P, t*s* e t*w* increase**
  - **Efficiency increases with N and Tc;**

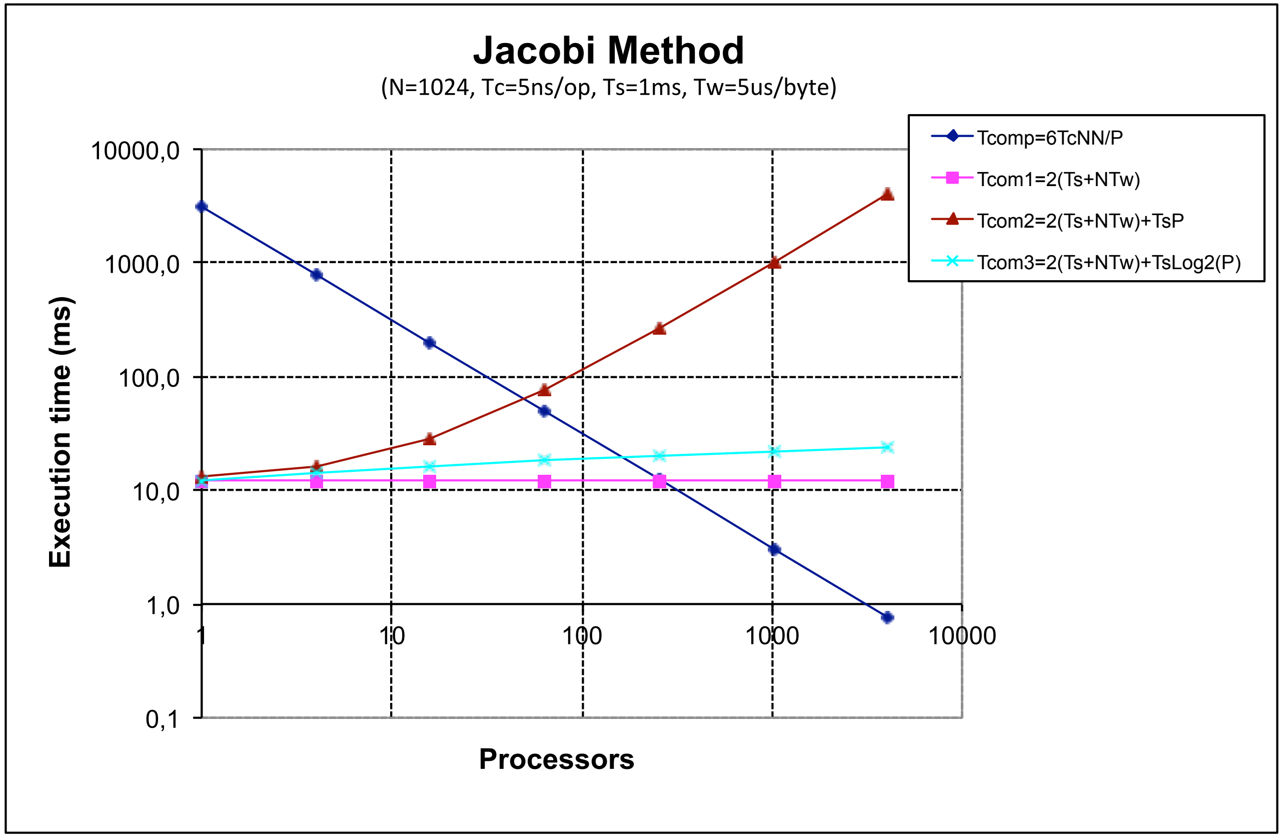
Texe = 6t*c*N²/P + 2t*s* + 2t*w*N

$$E = \frac{6t_cN^2}{6t_cN^2 + 2Pt_s + 2Pt_wN}$$

- *Scalability for problems with fixed size*.
  - Analysis of T*exec* and *E* when P increases
  - In general, *E* decreases. T*exec* can increase if it has a positive power of P.
- *Scalability for problems with variable size*
  - In some cases, more processors are used to solve larger problems, keeping the same efficiency levels
  - Isoefficiency indicates what is the required increase in the problem dimension, to keep the same efficiency, when the number of processors increases

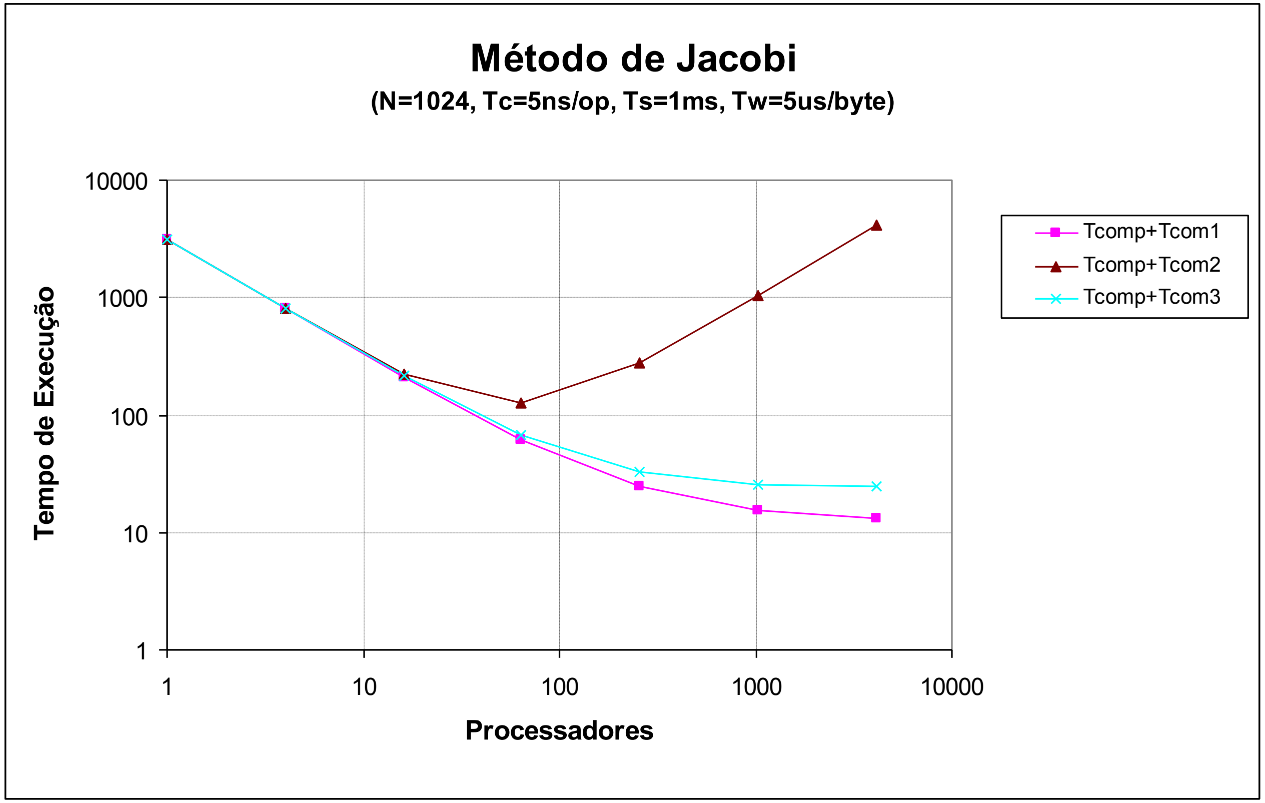
Performance of parallel applications

Scalability analysis (cont)



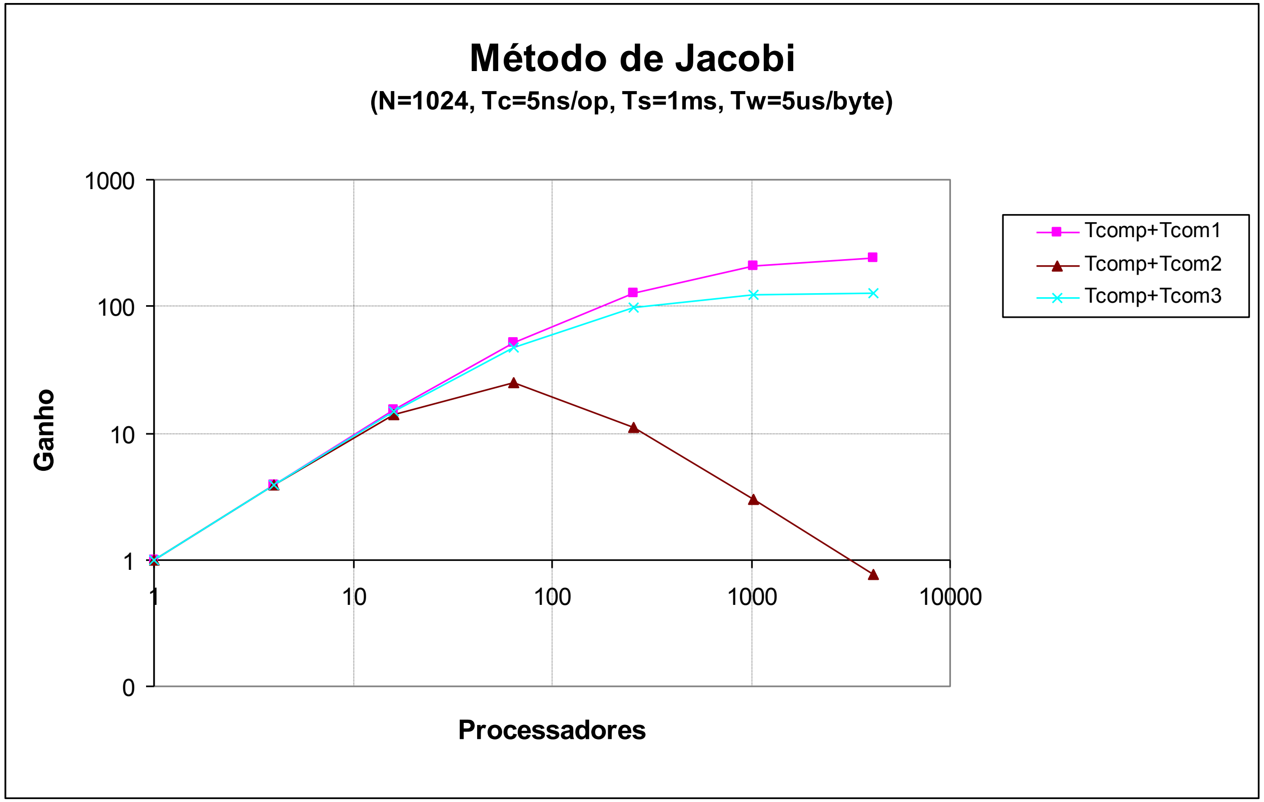
Performance of parallel applications

Scalability analysis (cont)



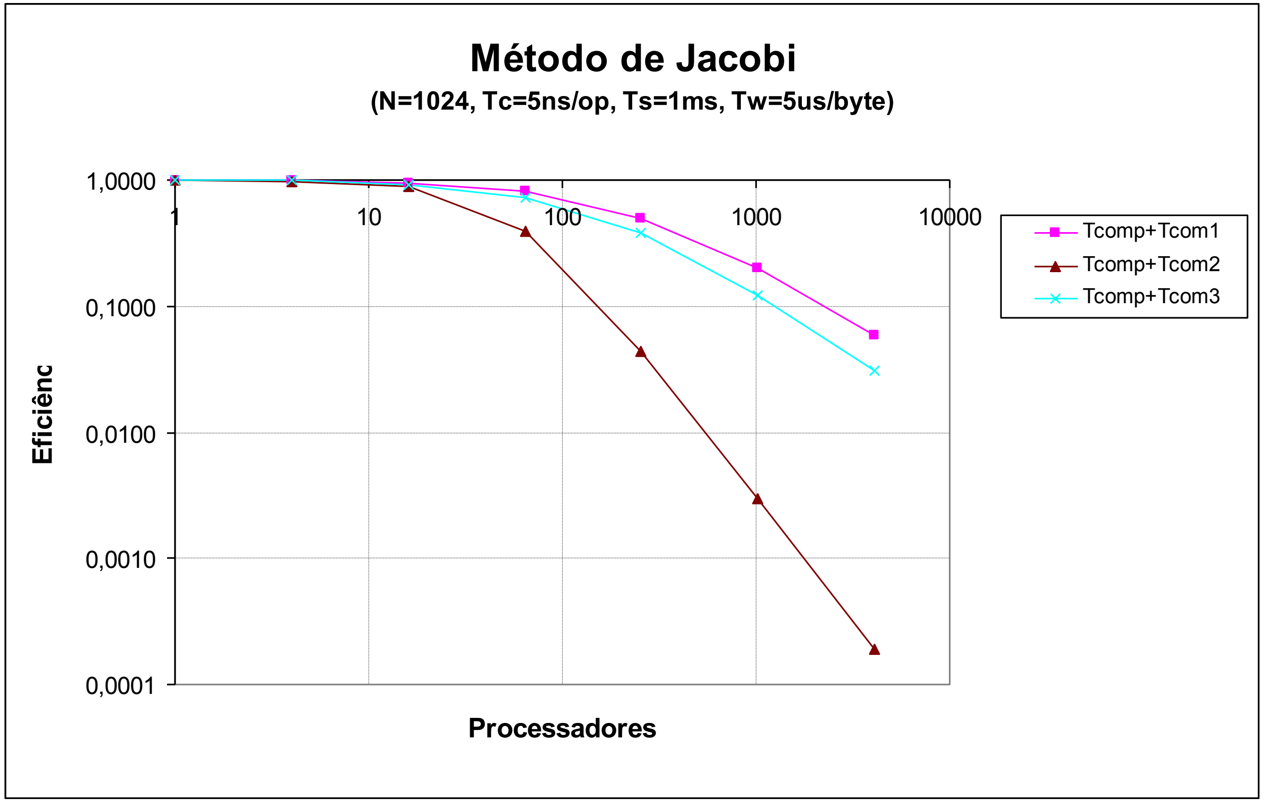
Performance of parallel applications

Scalability analysis (cont)



Performance of parallel applications

Scalability analysis (cont)



Performance of parallel applications

Measuring time in MPI

- **Time functions in MPI**
  - double MPI_Wtime() – returns the wall time (high resolution)
  - double MPI_Wtick() – returns the clock resolution (in seconds)

- **Wall time can differ from process to process**
  - There is no notion of “global time”
    - Each machine provides a local wall time
  - Application execution time should be wall time of the slowest process
  - Note: in some parallel algorithms process termination is not trivial

Performance of parallel applications

Experimental study and evaluation of implementations

- Parallel computing has a strong experimental component
  - Many problems are too complex for a realization only based on models
  - Performance model can be calibrated with experimental data (e.g., Tc)

- How to ensure that result are precise and reproducible?
  - Perform multiple experiments and verify clock resolution
  - Results should not change among in small difference: less than 2-3%

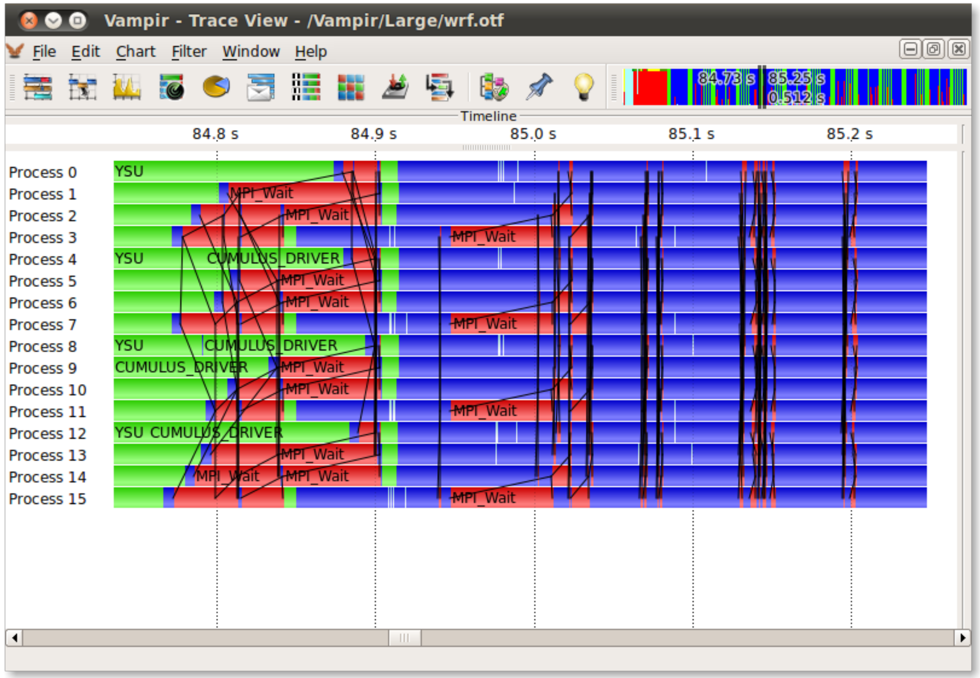
- Execution profile:
  - Gather several performance data: number of messages, data volume transmited
  - Can be implemented by specific tools or by directly instrumenting the code
    - There is always an overhead introduced in the base application

- *Speed-up anomalies*
  - superlinear (superior to the number of processors) – in most cases it is due the cache effect

Performance of parallel applications

Technique to measure the application time-profile (*profiling*)

- **Polling**: the application is periodically interrupted to collect performance data
- **Instrumentation**: code is introduced (by the programmer or by tools) to collect performance data about useful events
- Instrumentation tends do produce better results but also produces more interference (e.g., overhead)
- Exemplo: vampir



Performance of parallel applications

Distributed memory (MPI) vs Shared memory (OpenMP) optimisation

- **Distributed memory vs shared memory**
  - Data placement is explicit (vs implicit)
  - Static scheduling is preferred (vs dynamic)
  - Synchronization is costly (only by global barriers & message send)

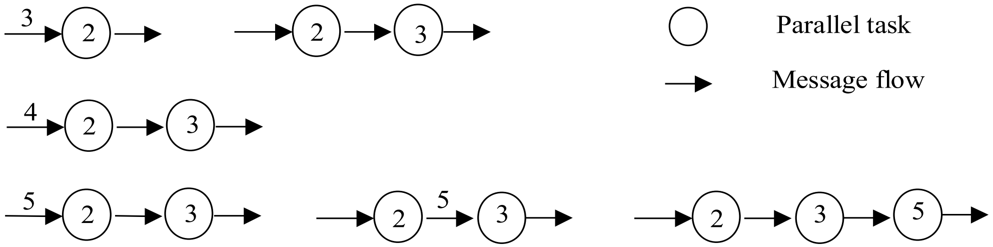
- Improve scalability on distributed memory
  - Minimise communication among processes
    - Eventually duplicating computation
  - Minimise idle time with a good load distribution

- Practical advise
  - Measure communication overhead
  - Measure load balance
  - Avoid centralised control

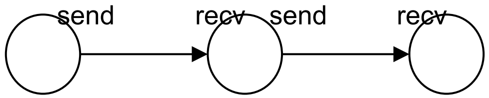
Exercício

Cálculo de números primos através do crivo de Eratosthenes

- algoritmo para calcular todos os primos até um determinado máximo
- pode ser implementado por uma cadeia de atividades, onde cada elemento filtra os seus múltiplos
- **o**s números são enviados para a cadeia por ordem crescente. Cada elemento que chega ao fim da cadeia é primo e é acrescentado ao fim desta como um novo filtro.



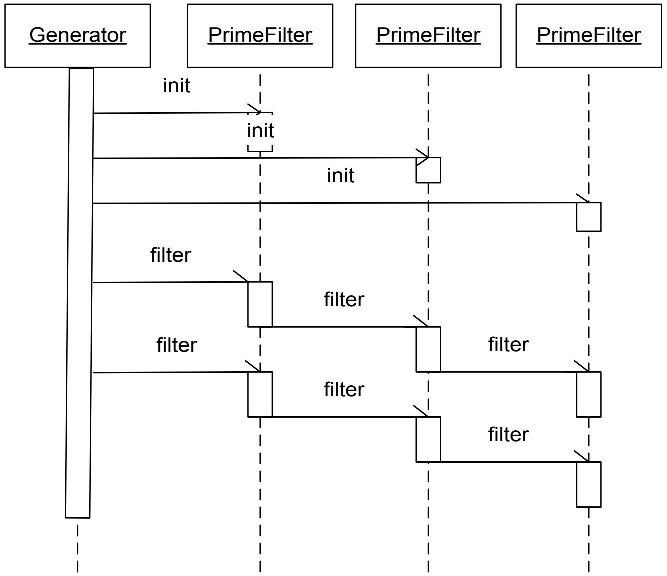
- atividade paralela tem um rácio entre computação e a comunicação de uma operação aritmética de inteiros (divisão) por mensagem
  - rácio demasiado baixo para a generalidade das plataformas de memória distribuída.

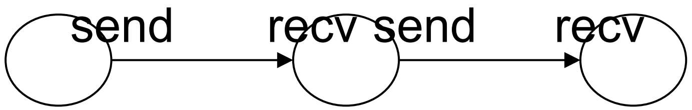


Passagem de Mensagens

Exemplo: Cálculo de números primos (RMI vs MPI)

- **JavaRMI** – O gerador invoca o método *filter* em cada filtro. O método *filter* é invocando entre filtros
- **MPI** – Os parâmetros de *init* são passados na linha de comandos (ou através de uma mensagem inicial). Os pacotes de números dever ser recebidos explicitamente e enviados ao filtro seguinte após o processamento.





Passagem de Mensagens

Exemplo: Cálculo de números primos (RMI vs MPI), cont.

- **Cadeia de três objetos/processos para calcular os números primos:**

```
int MAXP = 1000000;
int SMAXP = 1000;

PrimeServer *ps1 = new PrimeServer();
PrimeServer *ps2 = new PrimeServer();
PrimeServer *ps3 = new PrimeServer();

ps1->minitFilter(1,SMAXP/3,SMAXP);
ps2->minitFilter(SMAXP/3+1,2*SMAXP/3,SMAXP);
ps3->minitFilter(2*SMAXP/3+1,SMAXP,SMAXP);

int pack=MAXP/10;
int *ar = new int[pack/2];
for(int i=0; i<10; i++) {
    generate(i*pack, (i+1)*pack, ar);
    ps1->mprocess(ar,pack/2);
    ps2->mprocess(ar,pack/2);
    ps3->mprocess(ar,pack/2);
}
ps3->end();
```

```
int myrank = comm.rank();
…
   if (myrank==0) {
                  … // criar e iniciar filtro local
                  … // gerar pacotes de números
                  … // processar
                  comm.send(…);
   } else if(myrank==1) {
                  … // criar e iniciar filtro local
                  comm.recv(…);
                  …// processar
                  comm.send(…);
   else {
                  … // criar e iniciar filtro local
                  comm.recv(…);
                  …// processar
   }
```

Passagem de Mensagens

Exercícios

- **Alterar o código anterior para implementar uma *Pipeline***
  - Optimizar a implementação para melhor balanceamento da carga
- **Alterar o código para implementar um *farming*.**