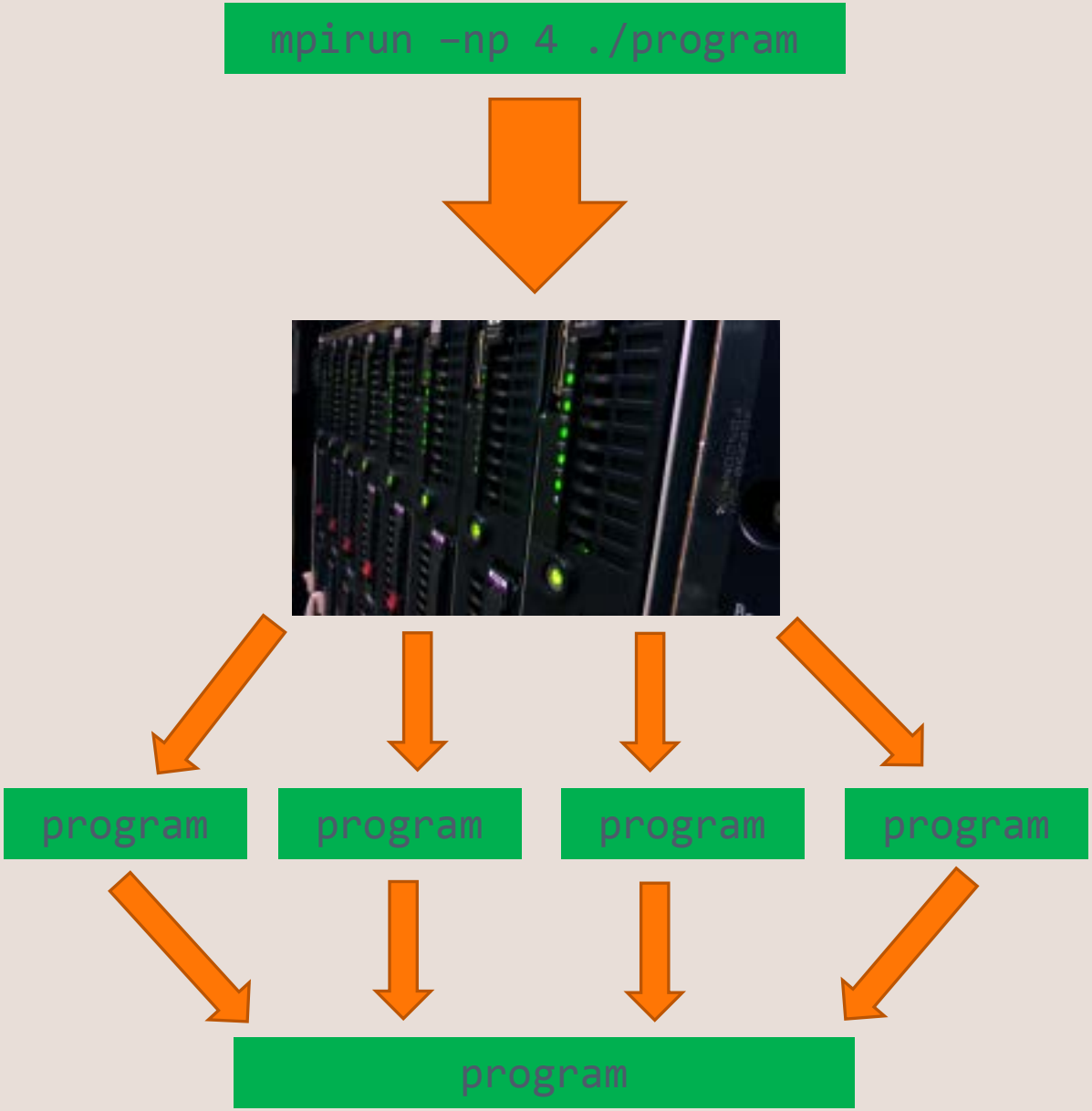
CRIDES ASÍNCRONES EN MPI: MILLORANT TEMPS I ENERGIA

Òscar Camacho Requena

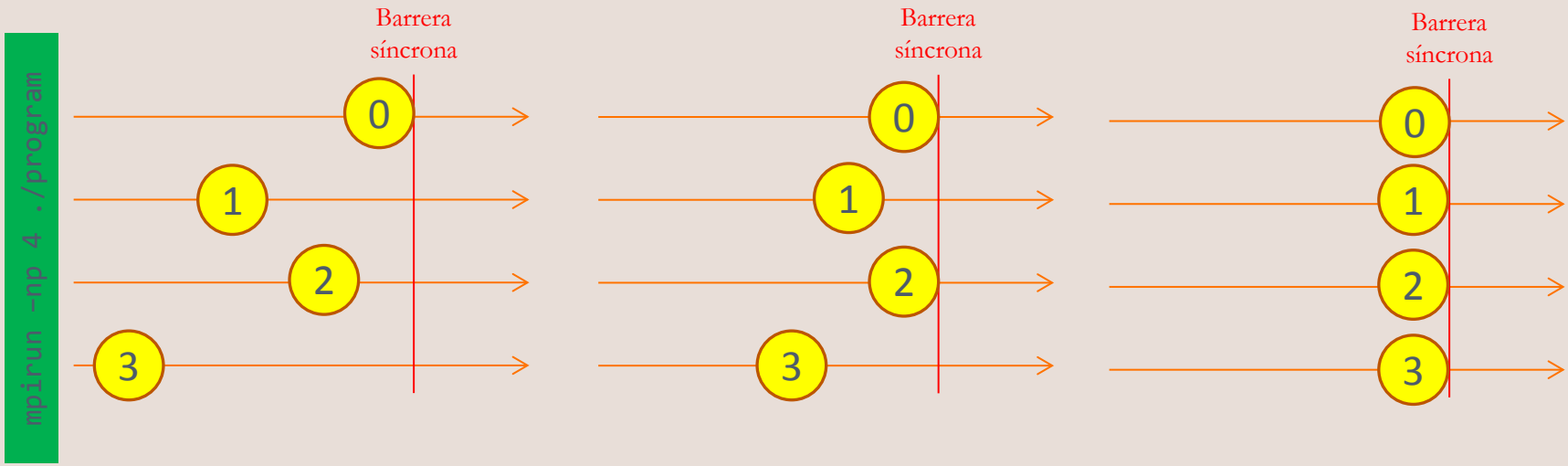
TFM – Computació Altes Prestacions

UOC - Juny 2015

MPI: llibreria de programació que permet realitzar fàcilment computació paral·lela d'altes prestacions en un clúster d'ordinadors.

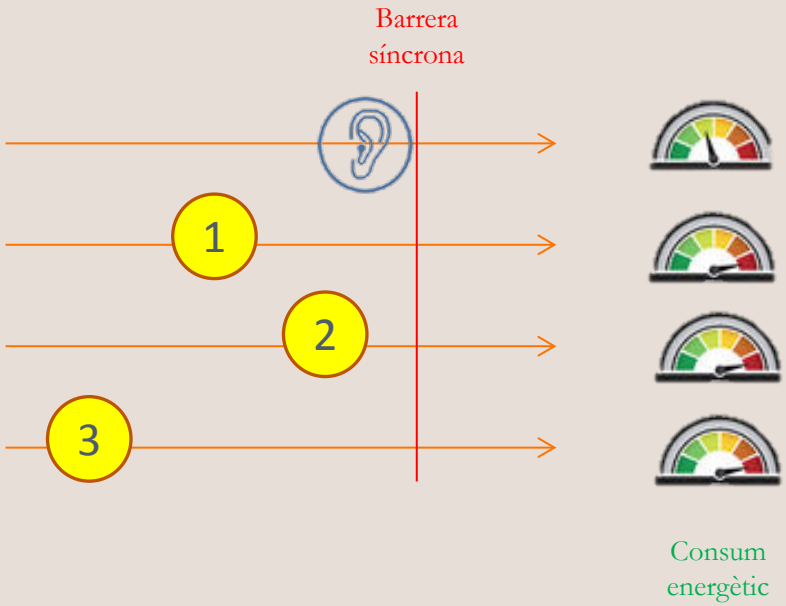


Per poder sincronitzar les operacions entre processos, MPI te una sèrie de crides col·lectives. Aquestes crides fan de barrera als fils d'execució.



De manera que fins que no arriben tots els fils d'execució a la crida, el programa no continua la seva execució.

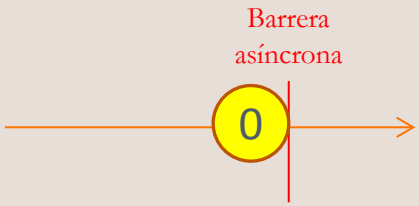
Aquest model d'execució pot arribar a ser ineficient, ja que els processos que arribin abans a la barrera, queden "escoltant" per saber quan han arribat a la barrera tots els fils d'execució i així poder continuar endavant.

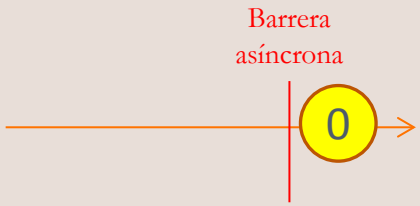


Aquesta espera activa, fa que el fil d'execució estigui consumint energia, encara que no estigui fent res.

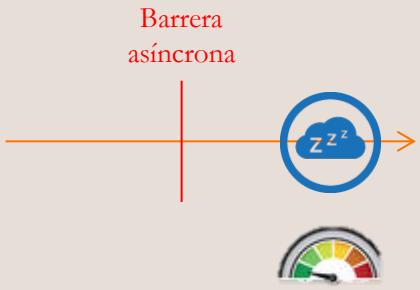
Per evitar aquestes situacions, fem servir les crides col·lectives asíncrones d'MPI.

Aquests tipus de crides no aturen els processos.





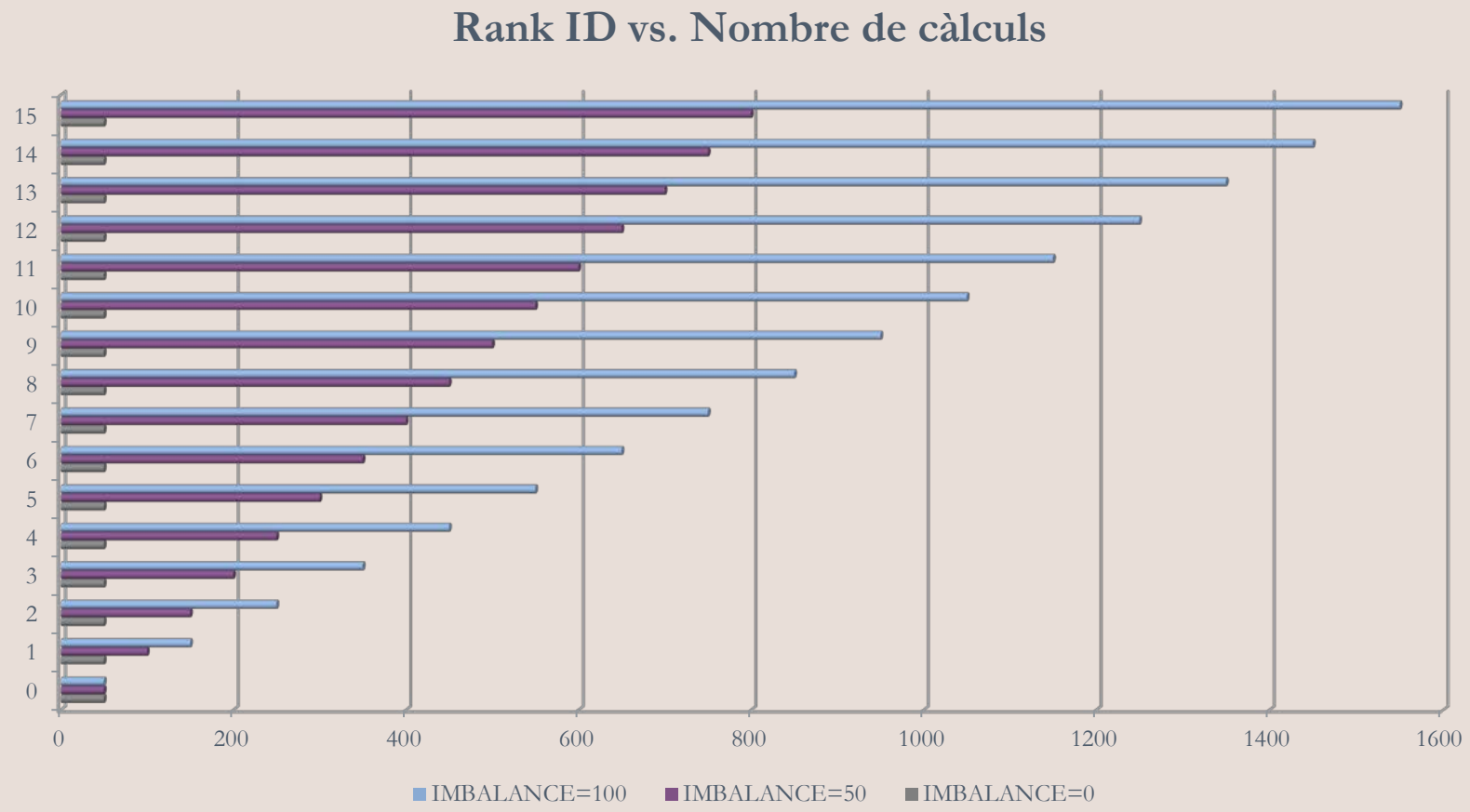
Si modifiquem el programa de manera que el fil d'execució entri en espera un temps abans de preguntar si la resta de fils han arribat a la barrera...



...aconseguirem disminuir el consum energètic del programa.

El primer programa utilitzat per realitzar la demostració,
és un programa de multiplicació de matrius.

Afegim un desequilibri en el nombre de càlculs que realitzen els fils d'execució,
per veure si afecta al rendiment del programa.



El programa incorpora una crida MPI_Barrier

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);


for(i=0; i<ITERS; i++){
      for(j=0; j<IMBALANCE_BASE+IMBALANCE*MyProc; j++){
              matmul(SIZE);
      }
      MPI_Barrier(MPI_COMM_WORLD);
      if(MyProc == 0){
              printf("Iteration %d done!\n", i);
      }
}

MPI_Finalize();
```

SÍNCRON

ASÍNCRON

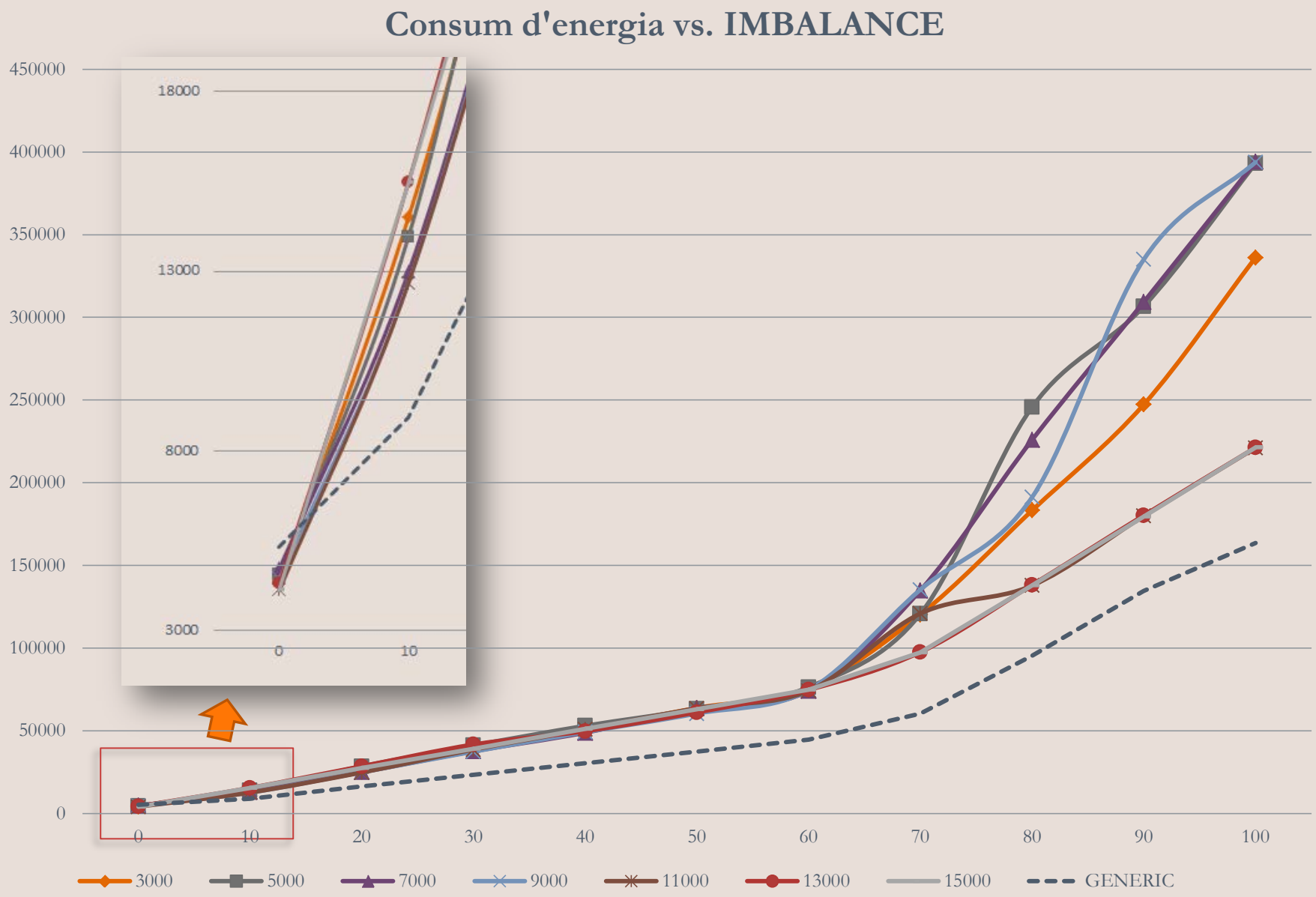
```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);

      if (MyProc == 0)
      {
printf("nombre arguments %d\n", argc);
printf("nsec %d\n", req.tv_nsec);
printf("SIZE %d\n", SIZE);
printf("IMBALANCE %d\n", IMBALANCE);
printf("ITERS %d\n", ITERS);
      }


for(i=0; i<ITERS; i++){
      for(j=0; j<IMBALANCE_BASE+IMBALANCE*MyProc; j++){
              matmul(SIZE);
      }

      MPI_Ibarrier(MPI_COMM_WORLD,&request);
      MPI_Test(&request, &flag, &status);
      while(!flag){
      clock_nanosleep(CLOCK, 0, &req, NULL);
      MPI_Test(&request, &flag, &status);
      }
      //MPI_Barrier(MPI_COMM_WORLD);


      if(MyProc == 0){
              printf("Iteration %d done!\n", i);
      }
}

MPI_Finalize();
```

que transformarem en MPI_Ibarrier per tenir la versió asíncrona del programa.

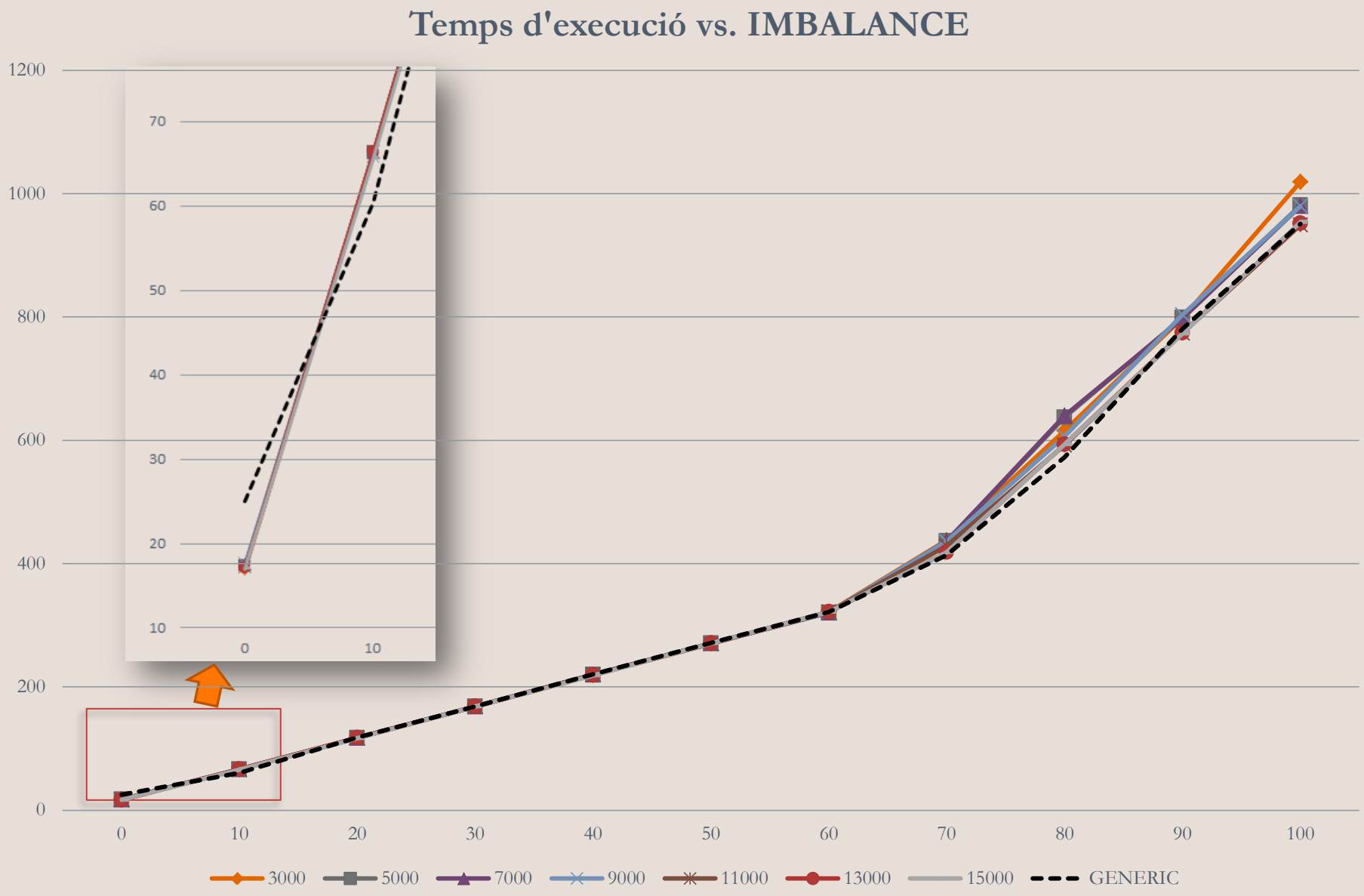
També incorporem el mecanisme necessari per sincronitzar els processos de manera asíncrona:

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &MyProc);

      if (MyProc == 0)
      {
printf("nombre arguments %d\n", argc);
printf("nsec %d\n", req.tv_nsec);
printf("SIZE %d\n", SIZE);
printf("IMBALANCE %d\n", IMBALANCE);
printf("ITERS %d\n", ITERS);
      }


for(i=0; i<ITERS; i++){
      for(j=0; j<IMBALANCE_BASE+IMBALANCE*MyProc; j++){
              matmul(SIZE);
      }

      MPI_Ibarrier(MPI_COMM_WORLD,&request);
      MPI_Test(&request, &flag, &status);
      while(!flag){
      clock_nanosleep(CLOCK, 0, &req, NULL);
      MPI_Test(&request, &flag, &status);
      }
      //MPI_Barrier(MPI_COMM_WORLD);


      if(MyProc == 0){
              printf("Iteration %d done!\n", i);
      }
}

MPI_Finalize();
```

Resultats obtinguts



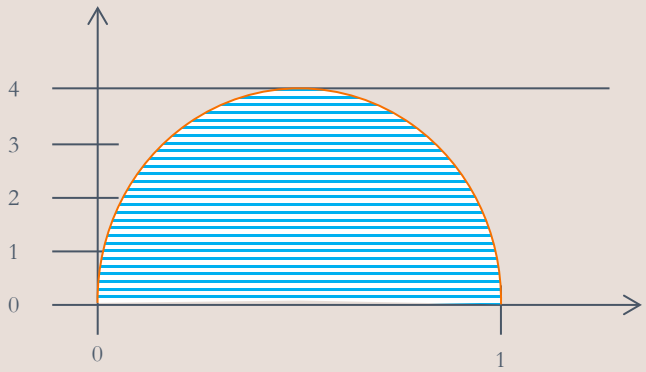
Resultats obtinguts



Un segon programa, que calcula el número π, ens servirà per fer proves amb una altra crida col·lectiva.

Utilitzem el teorema de Riemann per calcular la integral:

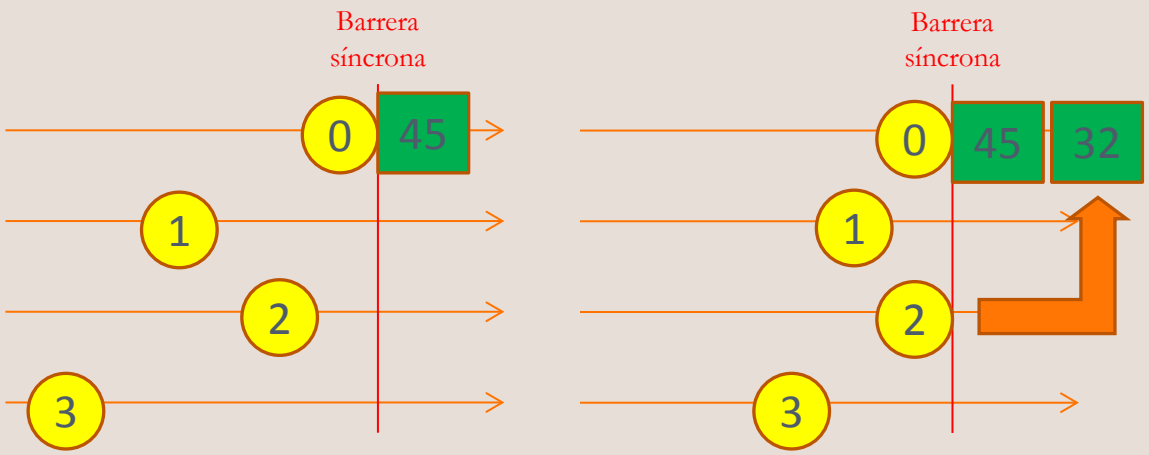
$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx$$



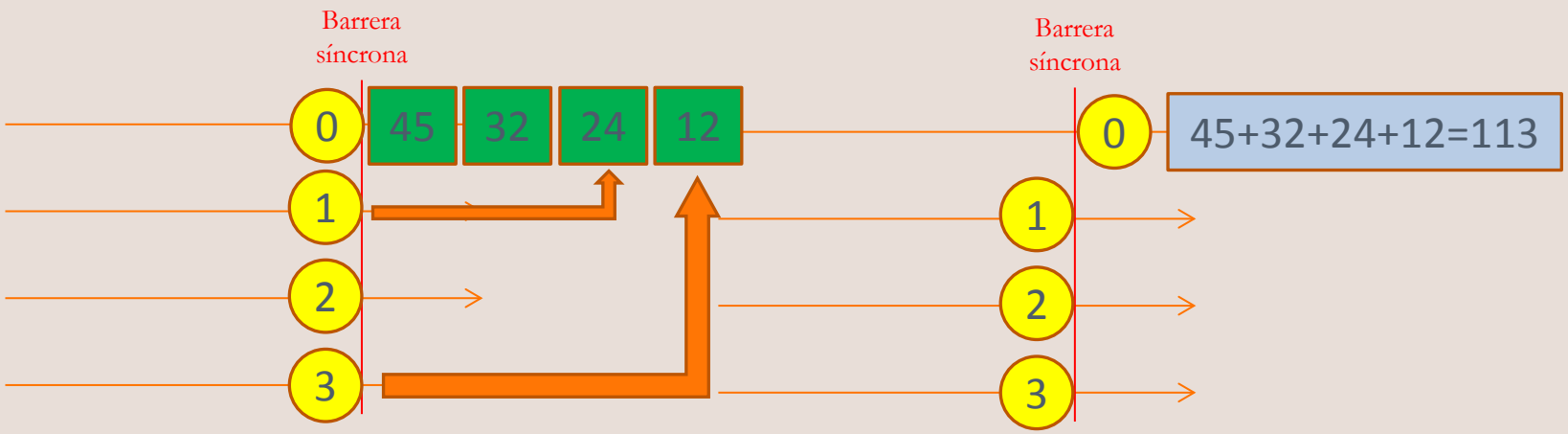
En aquest cas, farem servir les crides MPI_Reduce i MPI_Ireduce

Com funciona la crida col·lectiva MPI_Reduce?

Cada fil d'execució realitza un càlcul del qual obté un valor.



En arribar a la barrera, s'envia el resultat al procés principal, que realitzarà el càlcul final.



Transformació de la crida MPI_Reduce

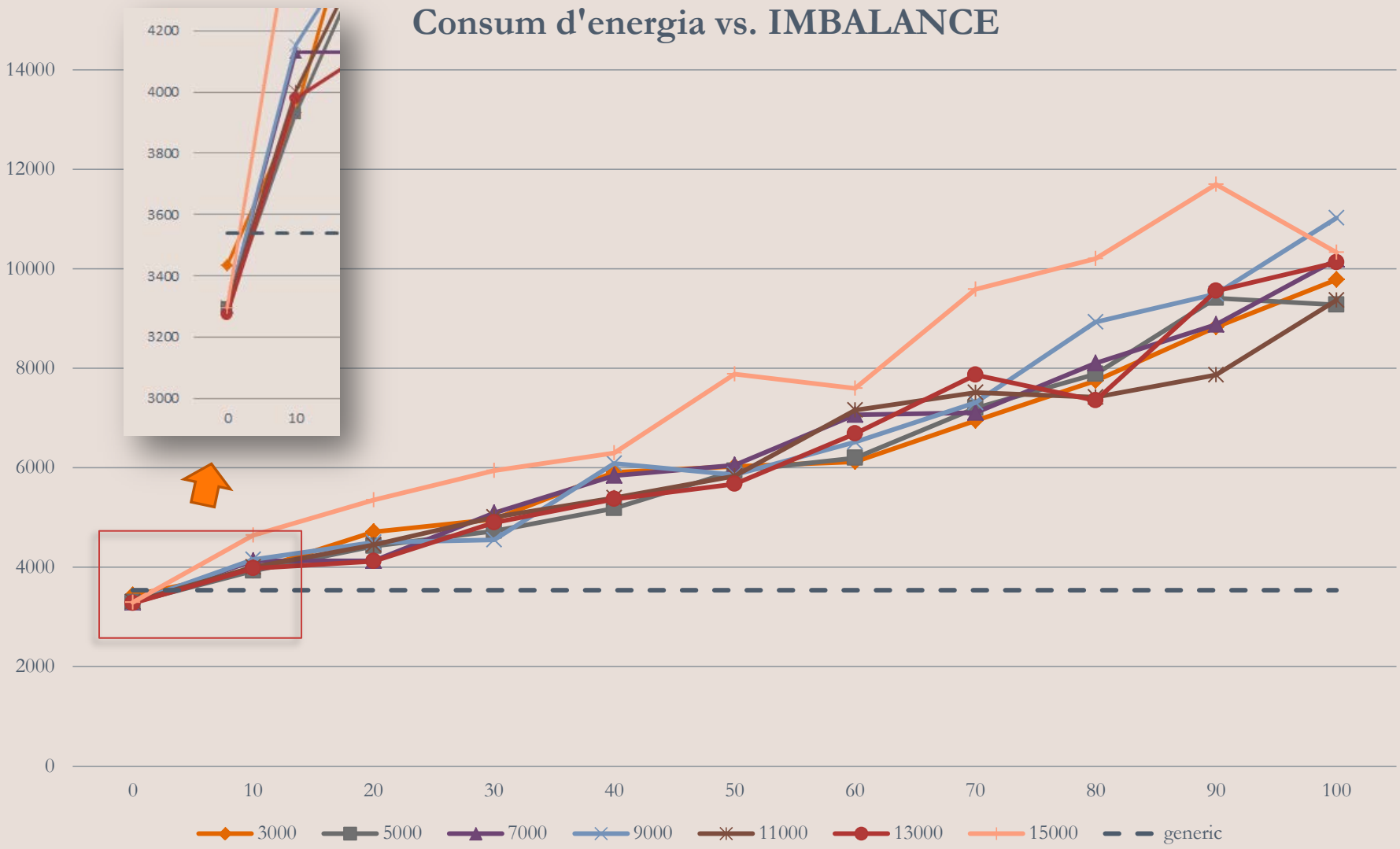
```
srand(myid);

for (i=0; i<mynpts; i++) {
  x = (double) rand()/RAND_MAX*(xmax-xmin) + xmin;
  mysum += 4.0/(1.0 + x*x);
}

MPI_Reduce(&mysum,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

if (myid == 0) {
  f = sum/npts;
  printf("PI calculated with %ld points = %f \n",npts,f);
}
MPI_Finalize();
```

SÍNCRON

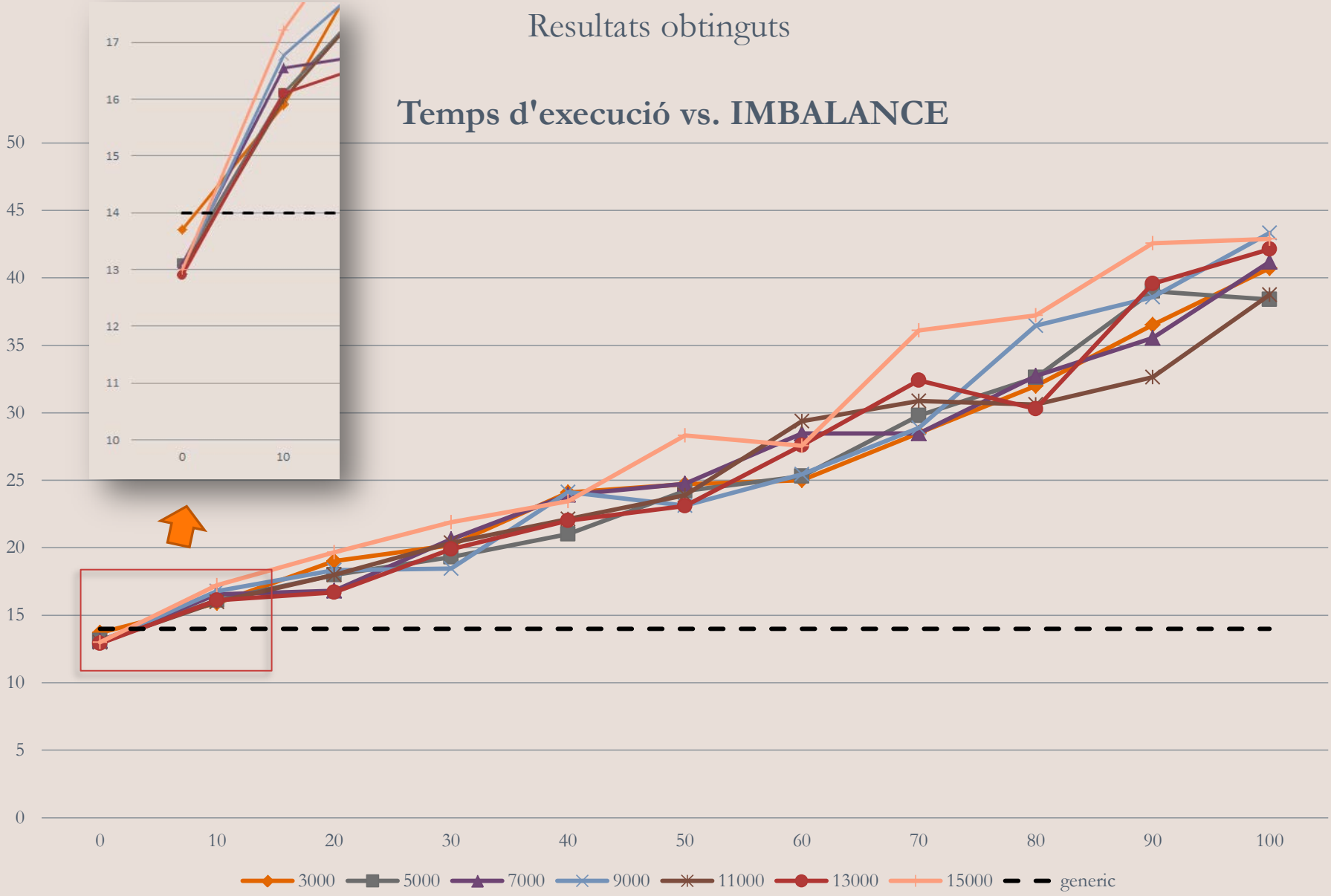
ASÍNCRON

```
srand(myid);
for (i=0; i<mynpts[myid]; i++) {
  x = (double) rand()/RAND_MAX*(xmax-xmin) + xmin;
  mysum += 4.0/(1.0 + x*x);
}
MPI_Ireduce(&mysum,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD,&request);
MPI_Test(&request, &flag, &status);
while (!flag)
{
  clock_nanosleep(CLOCK, 0, &req, NULL);
  MPI_Test(&request, &flag, &status);
}

if (myid == 0) {
  f = sum/npts;
  printf("PI calculated with %ld points = %f \n",npts,f);
}

MPI_Finalize();
```

Resultats obtinguts



Resultats obtinguts



CONCLUSIONS

- Les crides col·lectives asíncrones **tenen millor rendiment** que les síncrones, **milloren els temps d'execució** i **redueixen el consum energètic**.
- El desequilibri en els càlculs no millora els resultats.
- El temps d'espera asíncron és un factor actiu en la millora de resultats d'execució.

TREBALL FUTUR

- Optimització dels programes de proves.

  Fent-los realment asíncrons.

- Substitució en temps d'execució de les crides síncrones.

  Utilitzant les llibreries de *profiling* (pMPI)

- Optimització automàtica dels paràmetres asíncrons.

  Escollir automàticament els paràmetres idonis per obtenir els millors resultats de computació.

Òscar Camacho Requena

ocamacho@uoc.edu