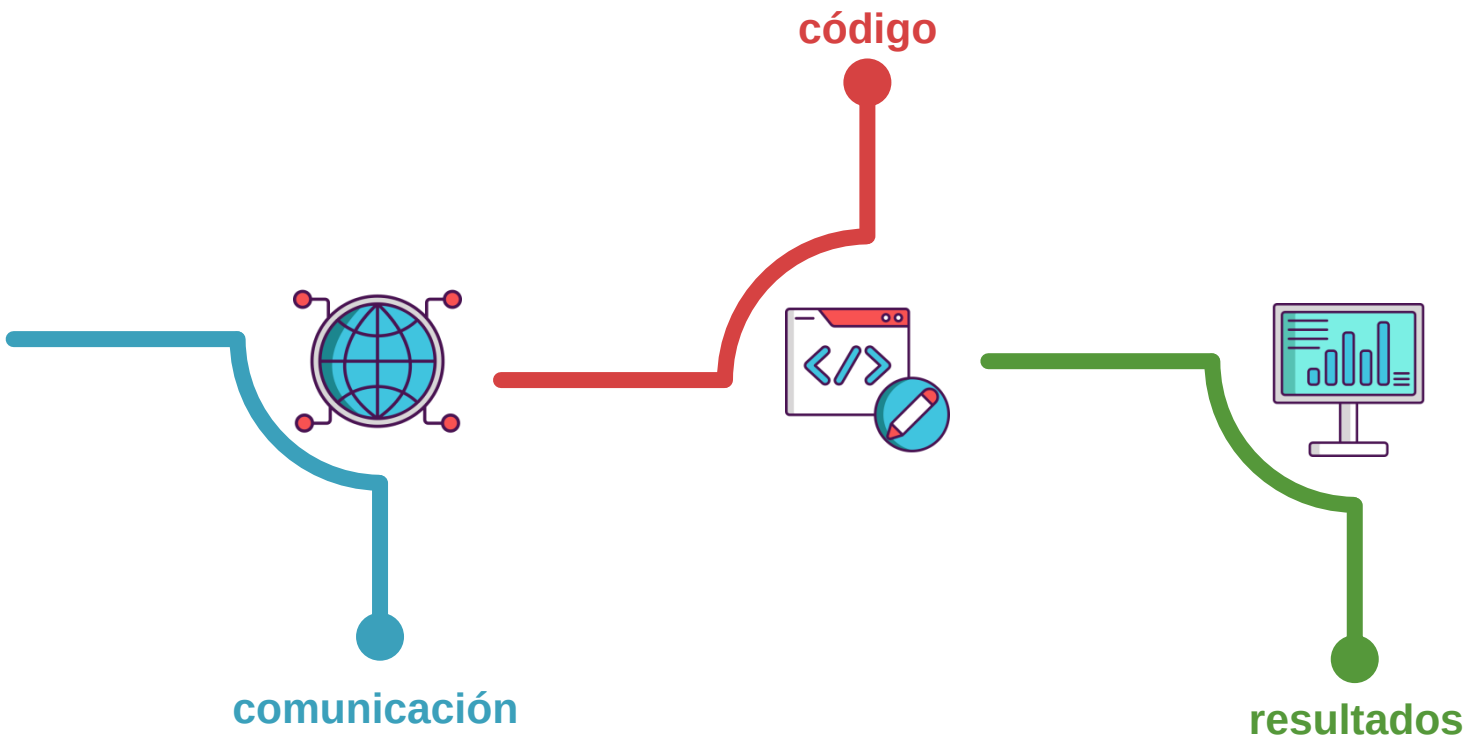
MPI

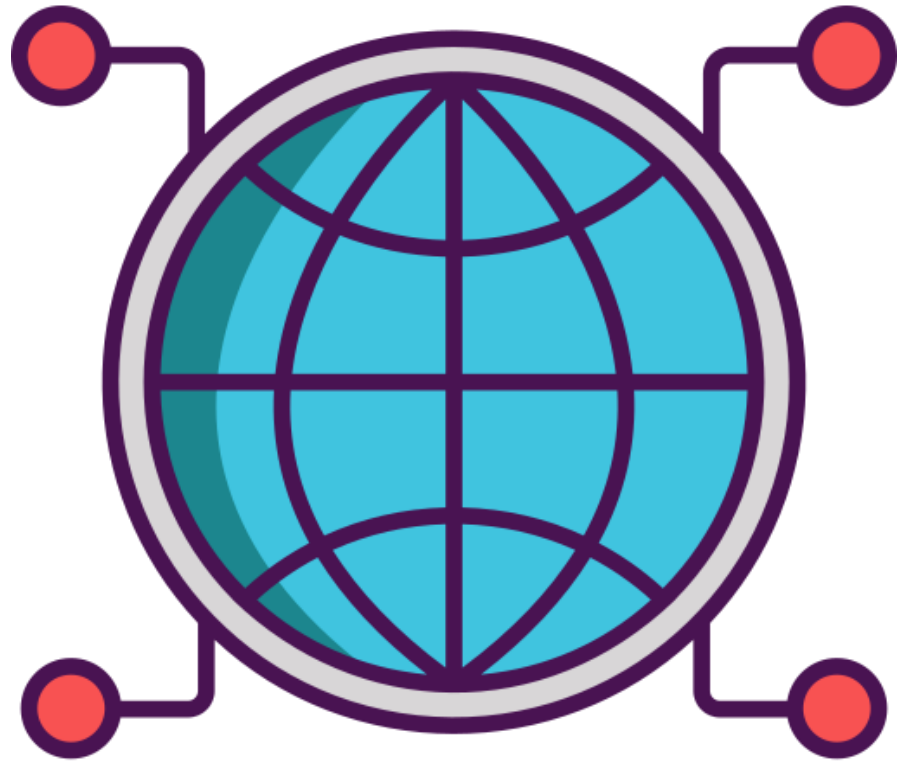
Álvaro Martín López

Francisco Pinto Santos

Héctor Sánchez San Blas

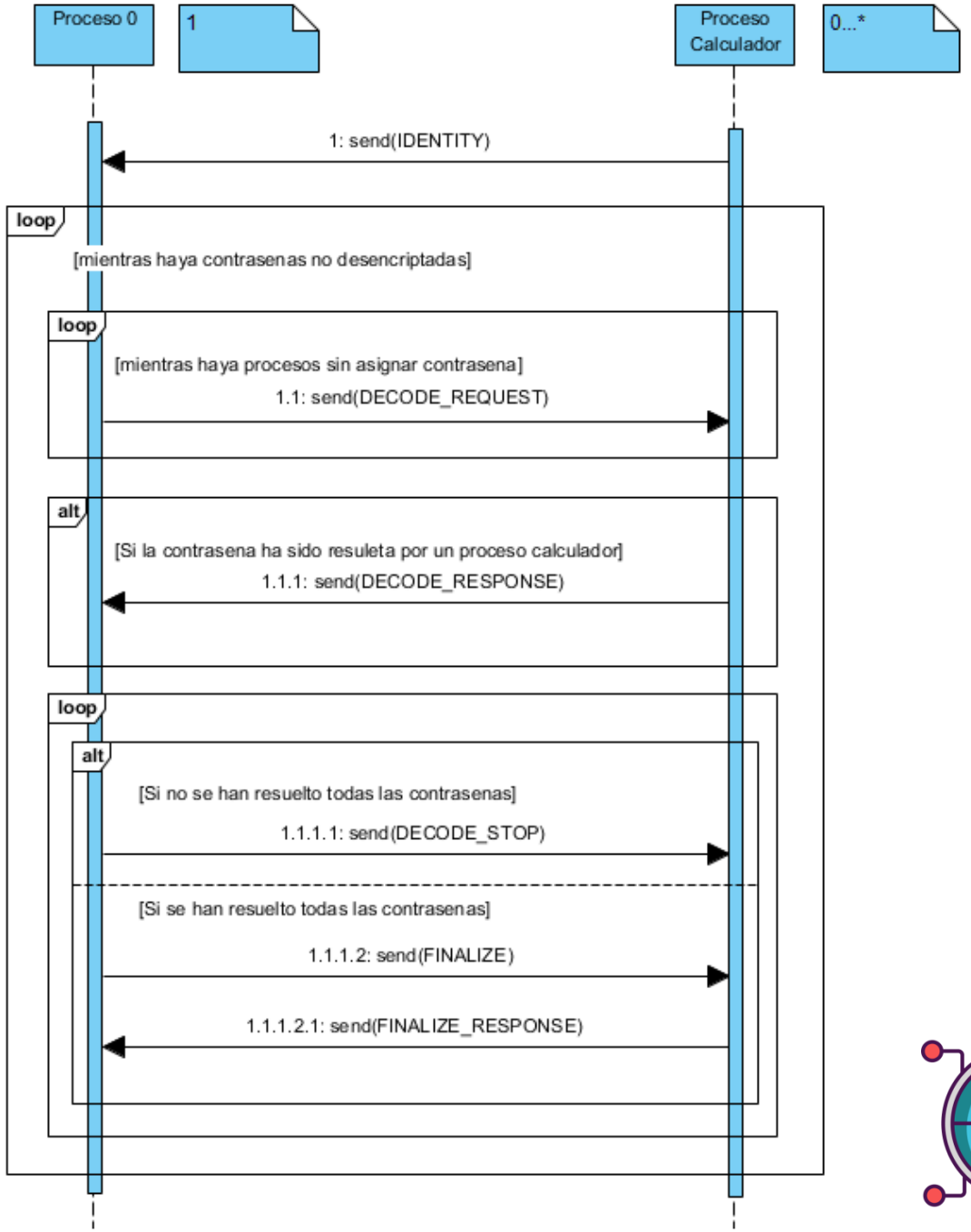
Manuel Salgado de la Iglesia


código
comunicación
resultados



Tipos de mensaje

- **IDENTITY**: un proceso calculador indica al proceso comunicador su PID, ID y máquina.
- **DECODE_REQUEST**: el proceso comunicador envía una petición de resolución de una contraseña en un rango.
- **DECODE_RESPONSE**: un proceso calculador indica que ha resuelto una contraseña.
- **DECODE_STOP**: el proceso comunicador indica que se debe parar de calcular.
- **FINALIZE**: el proceso comunicador indica que los cálculos han terminado y el proceso calculador deber morir.
- **FINALIZE_RESPONSE**: un proceso calculador le envía a el proceso comunicador datos usados para hacer las estadísticas.



Tipos de datos

- **Password**: datos de contraseña.
- **Request**: petición de intentar resolver una contraseña en el rango [rangeMin,rangeMax]
- **Response**: indicar que se ha resuelto la contraseña con ID passwordID, en ntries intentos y time segundos.
- **PasswordStatus**: estado de una contraseña y que tareas están trabajando en ella.
  Usado por el proceso comunicador para asignar tareas.

```
typedef struct{
    Salt s;
    PasswordID id;
    char decrypted[PASSWORD_SIZE];
    char encrypted[PASSWORD_SIZE];
}Password;

typedef struct{
    int rangeMin;
    int rangeMax;
    Password p;
}Request;

typedef struct{
    unsigned long long int ntries;
    double time;
    TaskID taskId;
    PasswordID passwordId;;
}Response;

typedef struct{
    bool finished;
    int numTasksDecrypting;

    PasswordID passwordId;
    TaskID taskIds[MAX_TASKS];

    Response solverResponse;
    double usedTime;
}PasswordStatus;
```

Comportamiento de proceso calculador

```
void calculationBehaviour(){
    bool solvedByMe;
    MessageTag tag;
    Request request;
    Response response;
    unsigned long long totalTries = 0;

    //give the seed
    srand(GET_SEED(ID));

    do{
        //Reset the request and response, and wait for a password request
        memset(&request,0,sizeof(Request));
        myRecv(MASTER_ID, 1, &request, MPI_REQUEST_STRUCT(request), DECODE_REQUEST);

        //Handle the request
        solvedByMe = requestHandler(request,&response);
        if(solvedByMe)
            mySend(MASTER_ID,1,&response,MPI_RESPONSE_STRUCT(response),DECODE_RESPONSE);
        //Add the number of tries
        totalTries += response.ntries;

        //Recive the decode_stop to discard it or finalize
        tag = myRecv(MASTER_ID, 0, NULL, NULL_DATATYPE, MPI_ANY_TAG);
        if(FINALIZE == tag){
            //send the total number of tries and finalize
            mySend(MASTER_ID, 1, &totalTries, MPI_UNSIGNED_LONG_LONG, FINALIZE_RESPONSE);
            return;
        }
    }while(TRUE);
}
```

Comportamiento del proceso comunicador

```
//generate passwords
for(i=0; i<N_PASSWORDS; i++){
    passwordList[i].id = i;
    GET_RANDMON_SALT(passwordList[i].s);
    GET_RANDOM_STR_IN_BOUNDS(passwordList[i].decrypted,0,MAX_RAND);
    ENCRYPT(passwordList[i].decrypted, passwordList[i].encrypted, passwordList[i].s);
}

//give the seed
//NOTE: the seed is given after the password generation, to ensure that the
//    communicator, have a different seed while the password decryption process
srand(GET_SEED(ID));

start = MPI_Wtime();

//wait for responses, and reasignate password to the free tasks
for(solvedPasswords = 0; solvedPasswords < N_PASSWORDS; solvedPasswords++){
    //division of tasks: if is the first time, the assignation is to all tasks, else is to the tasks implicated in the last solved password
    taskDispatcher(tasksToAssgin, nTasksToAssign, passwordList, passwordStatusList, &reqToMaster);

    //calculate the communicator's password and if the solver isn't the communicator, wait for a response
    solvedByMe = requestHandler(reqToMaster,&response);
    if(!solvedByMe)
        myRecv(MPI_ANY_SOURCE, 1, &response, MPI_RESPONSE_STRUCT(response), DECODE_RESPONSE);

    //save the end time, to have more accuracy on the total time
    end = MPI_Wtime();
    response.time = end - start;
    passwordStatusList[response.passwordId].usedTime = end - passwordStatusList[response.passwordId].usedTime;
    //add the response's tries to the total tries
    tmpTries += response.ntries;

    //stop all tasks working on the password except if it is the last time
    for(i=0; i<passwordStatusList[response.passwordId].numTasksDecrypting; i++){
        if(!IS_MASTER(passwordStatusList[response.passwordId].taskIds[i])){
            mySend(passwordStatusList[response.passwordId].taskIds[i], 0 , NULL, NULL_DATATYPE, (solvedPasswords!=N_PASSWORDS-1) ? DECODE_STOP : FINALIZE);
        }
    }

    //refresh the password status structures
    responseHandler(passwordStatusList,response);

    //save the tasks which were working on in the last password to assign them a new one in the next iteration
    tasksToAssgin = passwordStatusList[response.passwordId].taskIds;
    nTasksToAssign = passwordStatusList[response.passwordId].numTasksDecrypting;
}
```

Reparto de tareas

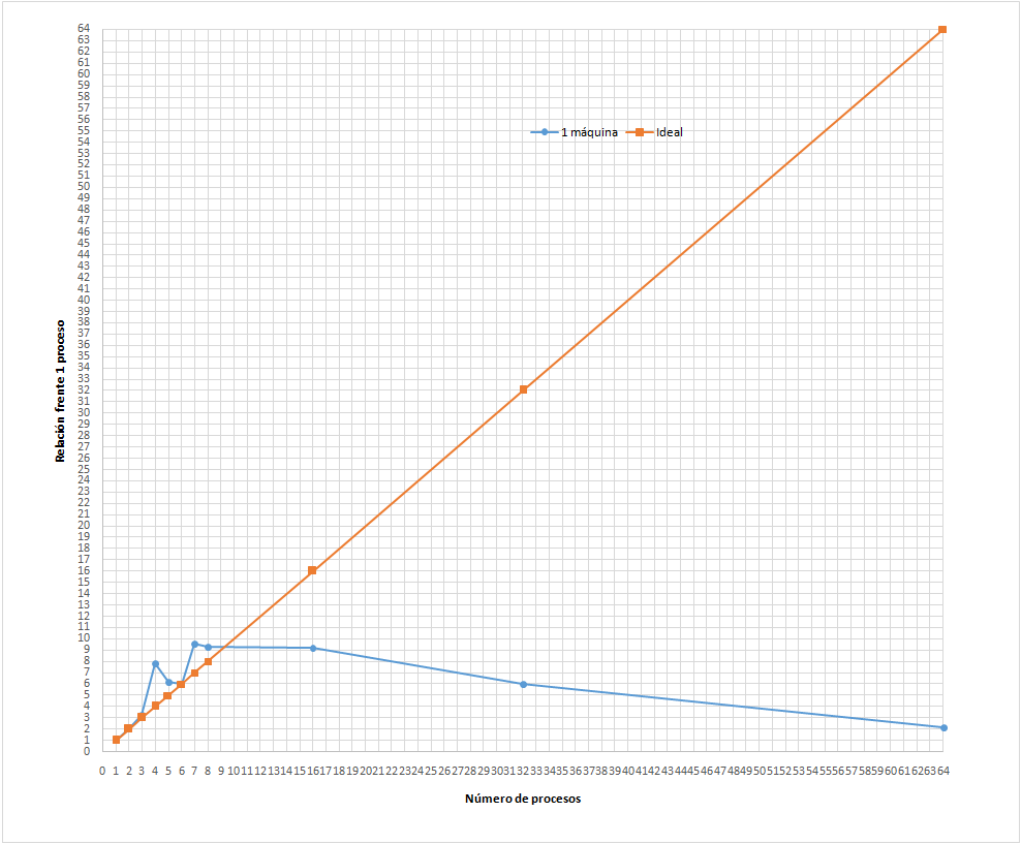
Política: asignar siempre a cada tarea la contraseña en la que menos procesos hay trabajando.
Dando prioridad a las contraseñas de ID mas bajo.

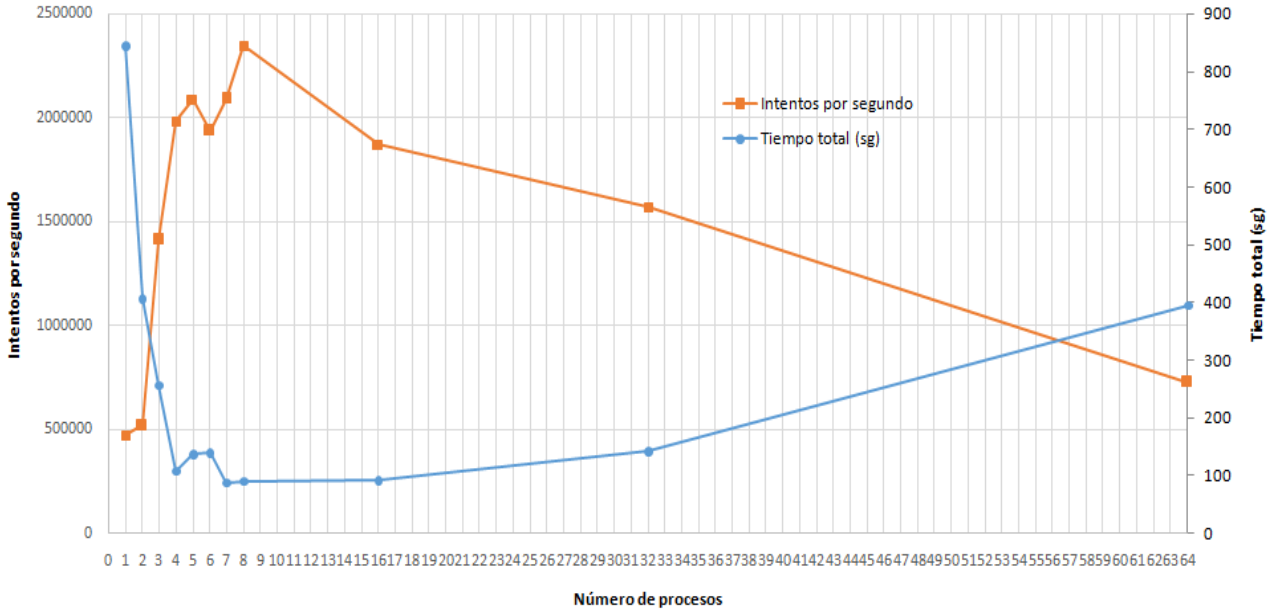
```
//assignation
for(currentTask=0; currentTask < nTasksToAssign; currentTask++){
    //search the password with the smallest number of tasks working on it
    selectedPassword = 0;
    for(currentPassword=0; currentPassword<N_PASSWORDS; currentPassword++){
        if( (FALSE==passwordStatusList[currentPassword].finished && passwordStatusList[currentPassword].numTasksDecrypting < passwordStatusList[selectedPassword].numTasksDecrypting)
        || TRUE == passwordStatusList[selectedPassword].finished ){
            selectedPassword = currentPassword;
        }
    }

    //assign the password to the task
    passwordAssignedToTask[currentTask] = selectedPassword;

    //stop the current processes working on the assigned password
    if(!hasBeenStopped[selectedPassword]){
        hasBeenStopped[selectedPassword] = TRUE;
        for(i=0; i<passwordStatusList[selectedPassword].numTasksDecrypting; i++){
            if(!IS_MASTER(passwordStatusList[selectedPassword].taskIds[i]))
                mySend(passwordStatusList[selectedPassword].taskIds[i], 0, NULL, NULL_DATATYPE, DECODE_STOP);
        }
    }

    //update the selected password status
    if(0 == passwordStatusList[selectedPassword].numTasksDecrypting){
        passwordStatusList[selectedPassword].usedTime = MPI_Wtime();
    }
    passwordStatusList[selectedPassword].taskIds[passwordStatusList[selectedPassword].numTasksDecrypting] = taskToAssign[currentTask];
    passwordStatusList[selectedPassword].numTasksDecrypting++;
}
```

Gestión de cálculos

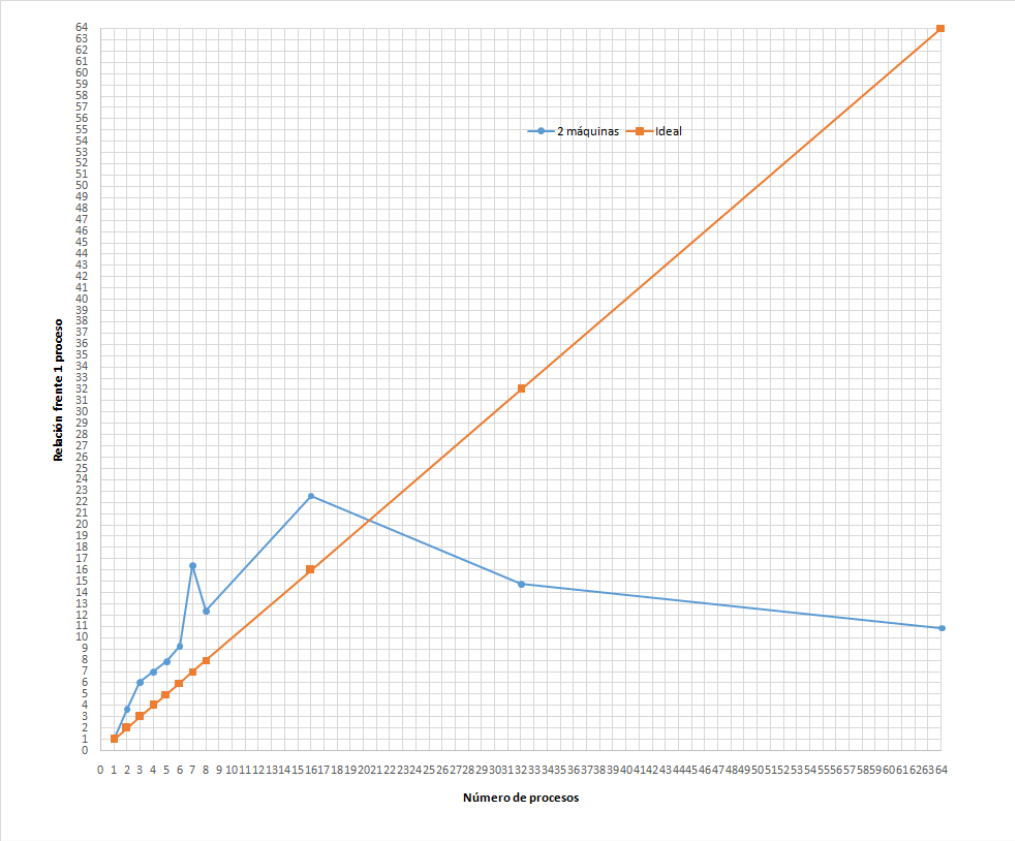
```
bool requestHandler(Request request, Response * response){
    long counter = 0;

    //Loop until password solved or a new response received
    do{
        //increment the try number
        counter++;

        //do the calculus and then check if the solution has been found
        if(doCalculus(&(request.p),request.rangeMin,request.rangeMax)){
            memset(response,0,sizeof(Response));
            //fill the response
            response->taskId = ID;
            response->ntries = counter;
            response->passwordId = request.p.id;
            //return TRUE, to know after if the password has been solved by me
            return TRUE;
        }

        //Check if a message has been received each NUMCHECKSMAIL times
        if( 0 == (counter % NUMCHECKSMAIL)){
            if(areThereAnyMsg()){
                response->ntries = counter;
                return FALSE;
            }
        }

    }while(TRUE);

}
```

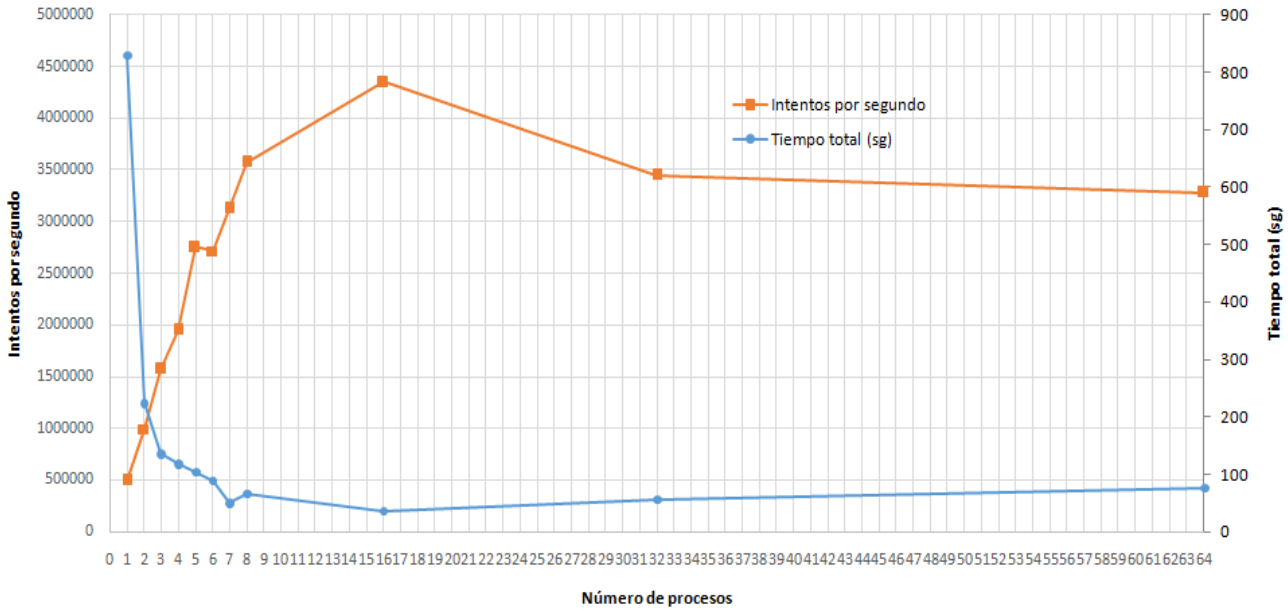
Número de maquinas: 1



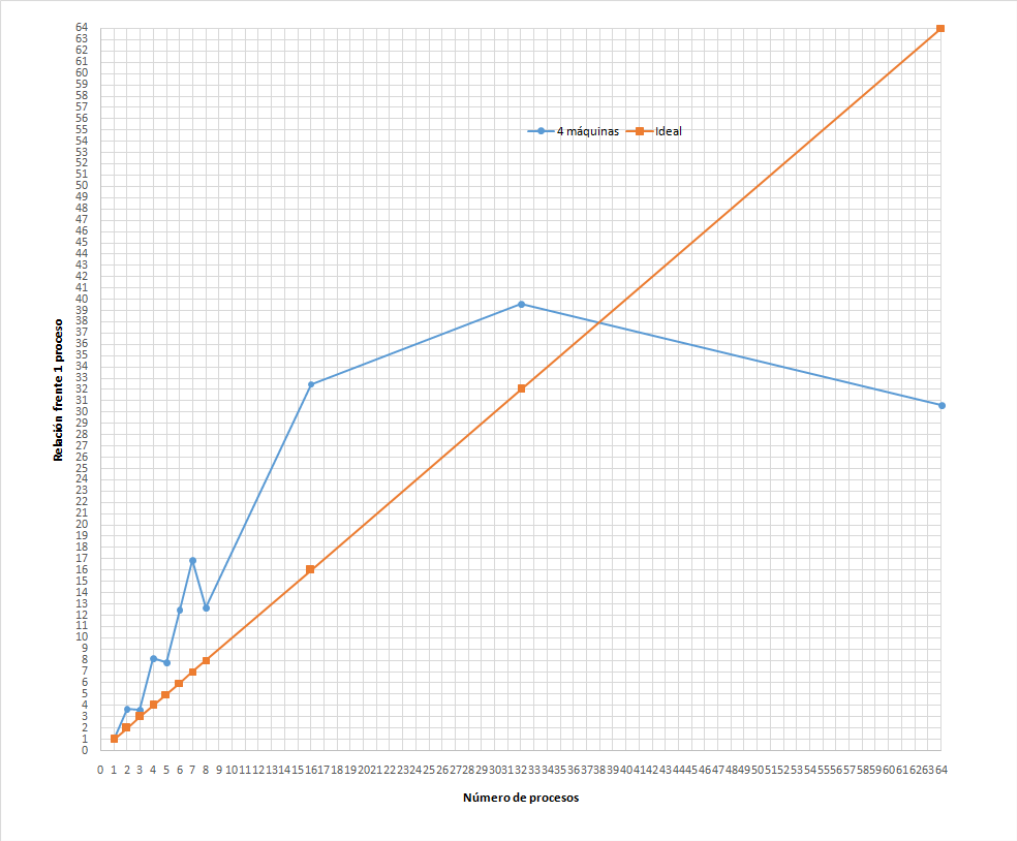


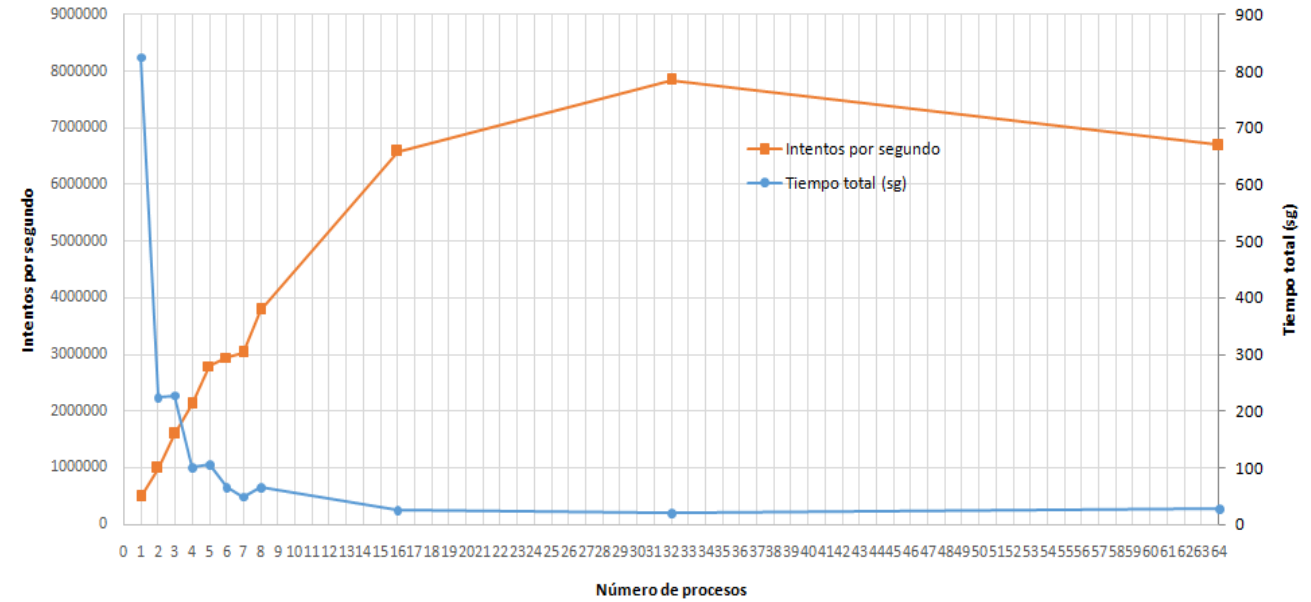
Número de maquinas: 2





Número de maquinas: 4





Conclusiones

- A medida que **aumenta** el número de **procesos**, **disminuye** el **tiempo** total
  - Se realiza el mismo trabajo, pero de forma paralela
  - Más procesos trabajando en una contraseña → rangos de búsqueda mas pequeños → más probable dar con la combinación correcta.

- A medida que **aumenta** el número de **procesos**, **aumenta** el número de **intentos/tiempo**
  - Se hacen intentos de forma paralela, por lo tanto es algo natural

- Estas características se verifican **hasta** que el **número** de **procesos** supera el numero de **núcleos** disponibles. Tras esto, el **rendimiento disminuye**
  - Aumenta el tiempo total, y se disminuyen los intentos/tiempo
  - El SO reparte de manera equitativa el tiempo de CPU entre ellos → compiten entre ellos por la CPU.

Conclusiones

- La **disminución** del **rendimiento** de los casos anteriores, al haber mas procesos que procesadores, se aprecia **menos** cuantas **más máquinas** haya
  - Al haber más núcleos, los procesos tienen que competir menos por ellos

- El **coste** de **comunicación** es **mayor** que el **beneficio** que aporta **si** hay **pocos procesos**
  - Cuando se utilizan varios procesos y hay pocas máquinas, si el número de procesos es pequeño, el rendimiento es peor que en el caso de una sola máquina
  - Los procesos se reparten de forma uniforme entre todas las maquinas, y el coste de la comunicación es mayor que el beneficio que aporta

- Estas **mediciones no** dan resultados **ideales** (formas logarítmicas exactas) porque las **mediciones no** son **exactas**
  - A pesar de tener 8 núcleos, cada máquina esta corriendo otros procesos como el SO, la terminal, el gestor gráfico de ventanas, …. En general, hay muchas variables que no controlamos porque el reparto de CPU depende del SO
  - No todas las CPU en ese momento son exactamente iguales: variaciones en el proceso de fabricación, distinto uso, distintas temperaturas ….
  - Para obtener resultados que se acerquen a lo ideal hace falta repetir cada experimento varias veces