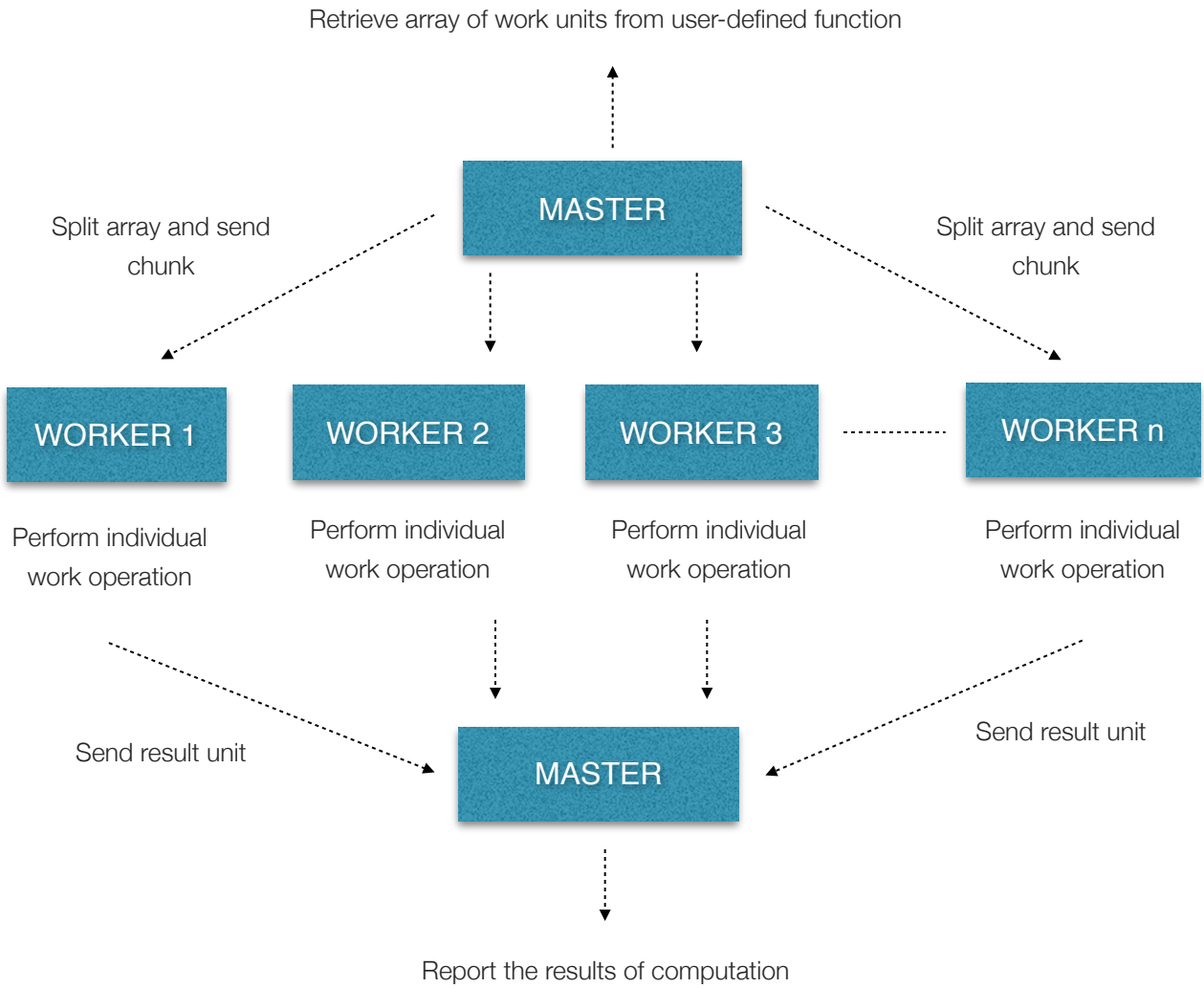
# Lab 2 - Report

## 1. MASTER - WORKER API

We have implemented a basic API for parallel programs in a master worker structure using MPI. It can be found in the file *api.h* and *api.c*. Please find the instructions of how to use it below. Also note that it is necessary to have the MPI library installed and included in *api.c.* With this API the user is able to create a parallel program in a simpler manner, simply indicating the functions that a master process and the rest of the worker processes must perform and the structure to handle the work units and result units.

The following figure represents the high level flow of the master-worker paradigm using our API. A more detailed definition of the functions and values follow:



This API has been tested using the *test.c* file. It generates a very simple work scheme which consists of 10 chunks with an array of size 10, each element of the array containing the value of its index. It then simply calculates the sum of the elements in each array and adds them all up.

## Data Structures and Functions

The data structures our API uses are the following:

- **one_work_t**: a *struct* of type *one_work* containing all the elements of work to be sent to an individual worker.
- **one_result_t**: a *struct* of type *one_result* containing all the elements of a "result" from the work of an individual worker that is returned to the master.
- **mw_fxns**: a *struct* that defines an instance of our API. It needs to be loaded in the main program executed by the user using the following **user-defined functions and parameters**:
    - *one_work_t **(*create_work_pool) (int arc, char **argv):* the function to be called by the "master" and that is in charge of dividing the work into several work chunks of type *one_work_t*. Returns a NULL-terminated list of one_work_t.
    - *one_result_t *(*do_one_work) (one_work_t* work)*: the function to be called by a worker. it receives a piece of work and does the execution the user defines on it. It returns the result in the form of *one_result_t*.
    - *int (*report_results) (int sz, one_result_t **result_array):* the function to be called by the master after collecting the results from every worker and storing them into an array. It reports the results in the way the user defines. Returns 0 on failure and 1 in success.
    - *int work_sz:* specifies the size in bytes of the one_work_t structure defined by the user.
    - *int result_sz:* specifies the size in bytes of the one_result_t structure defined by the user.
- **void MW_Run (int argc, char **argv, struct mw_fxns *f, int style)**: a *function* that performs the necessary executions using MPI to set up the master-worker structure and perform the necessary work. Please note that the *style* parameter can only be 1 or 2, indicating whether to use round-robin or dynamic processor selection.
    - Style = 1 is used for a basic round-robin: the master iterates over the work chunks and assigns each one a processor in order of processors available. It retrieves the results from the processors in a non-blocking manner after sending all the chunks, adds all the results to a single array, and then report the results using the *report_results* function.
    - Style =2 is used for a more advanced dynamic processor selection. The first *x* amount of work chunks are assigned to the *x* available processors by the master. Then the master waits for

results from any of these processors, and as soon as it receives one, it then sends the next work chunk to that process.

## How to use

In order to use our API please follow the instructions:

1. Load the file *api.h* into the main program.

2. Define the structures of the individual work and the result of a work performed by a worker:

```
struct one_work {
        ……
};
struct one_result {
        ……
};
```

3. Define the function *make_work, do_work, and report* necessary for the API to work:

```
one_work_t **make_work (int argc, char **argv){
        //Function to divide the work
        ……
}
one_result_t *do_work (one_work_t* work){
        //Function to the a unit of work and return a result
        ……
}
int report (int sz, one_result_t **result_array) {
        //Function to report the final result
        ……
}
```

*4.* In the *main* function, declare a new instance of our API, assign it all the parameters and functions previously specified, set up MPI_Initialize() and MPI_Finalize(), and run *MW_Run():*

```
int main (int argc, char **argv ){

    //Initialize our API

    struct mw_fxns mw;
    mw.create_work_pool = make_work;
    mw.do_one_work = do_work;
    mw.report_results = report;
    mw.work_sz = sizeof(one_work_t);
    mw.result_sz = sizeof(one_result_t);

    //IRun the program

    MPI_Init (&argc, &argv);
    MW_Run (argc, argv, &mw, 2);    //Runs the program in a dynamic processor selection style
    MPI_Finalize ();

}
```

*5.* To run the program, please run the following commands*:*

- `mpicc api.c "your_program".c -o "program_name"`
- `mpirun -np "number of processors" "program_name"`

## Important limitations

While we tried to develop our API as flexible as possible to meet the user's needs, there are certain limitations that must be considered:

1. The structure of *one_work* and *one_result* defined by the user **must not** contain any pointers. This is due to the fact the the workers will not be able to access the correct memory address to obtain the information stored in these pointers. A future improvement would be to incorporate a *serialize* and *deserialize* function for both the work chunks (one_work) and the results (one_result).

2. Because of this, the user must define in the code the number of elements a work chunk is going to have. All work units must contain the same sub elements of work (for example arrays to compute would need to be of the same size).

# 2. COMPUTING FACTORS OF A LARGE NUMBER

## Problem description

We are going to use our API to calculate the unique prime factors of a large number. In order to do this, we are going to iterate from 2 to the square root of the number and check if the number is prime and if the large number is divisible by it.

In order to this, we are using the *gmp* library because otherwise we would be limited by the maximum value of an integer for the large number, since the *sqrt* function from the *math.h* C library does not support longer data types such as *long int*. Please note that while the *gmp library* supports the possibility of having infinite large numbers, we have decided not to use that data type (*mpz_t*) and instead do our operations with a *long int*, using the math functions provided by *gmp* which do allow computations with such large numbers. The reason for this is that the size of a *mpz_t* not fixed, which brings about a lot of issues when handling memory allocation in the different workers that are receiving the work chunks and need to allocate memory for each.

## Problem solution

Using our API, we define the necessary structures in the *factor.c* file:

```
struct one_work {
    unsigned long potential_factors[WORK_ARRAY_SIZE];
};

struct one_result {
    unsigned long factors[WORK_ARRAY_SIZE];
};
```

Please note that WORK_ARRAY_SIZE is defined by the user and should be estimated based on the number he intends to calculate the prime factors for.

In the case of the functions, the basic functionality is the following:

- The *make_work* function is the one called by the master. It generates an array of work units that are to be sent to the worker processes. In this case, the amount of work units is determined by the square root of the large number in question, divided by the work array size defined by the user in the code. It then simply initializes a possible factor variable and iterates all the way up to the square root of the large number, assigning each value to an element of the possible_factors array in the corresponding work unit.

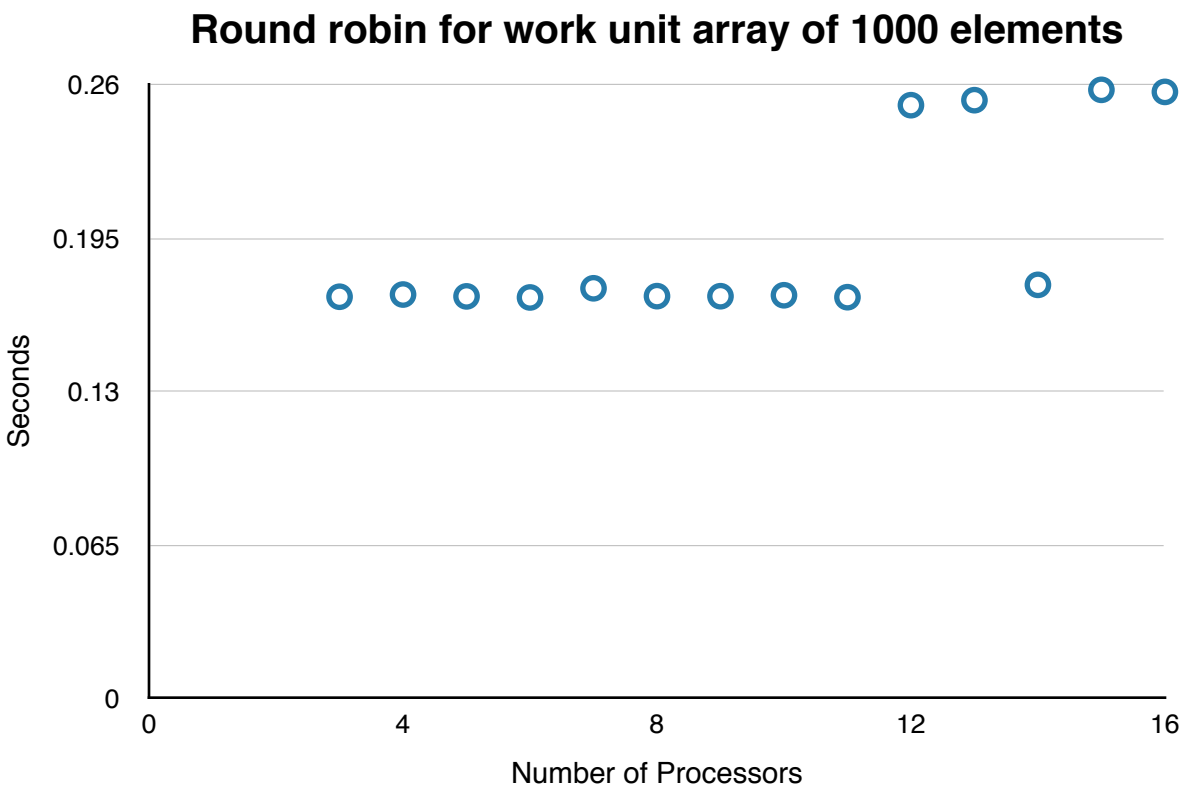
- The *do_work* simply receives a work unit, iterates over the possible factors array, and then checks if it is in fact a unique prime factor based on the explanation given above. It then stores the factors in the factors array of a result unit, and returns this result unit.

For a more complete description of the program, please look at the *factor.c* file. In order to run the program, please type the following commands in the terminal:

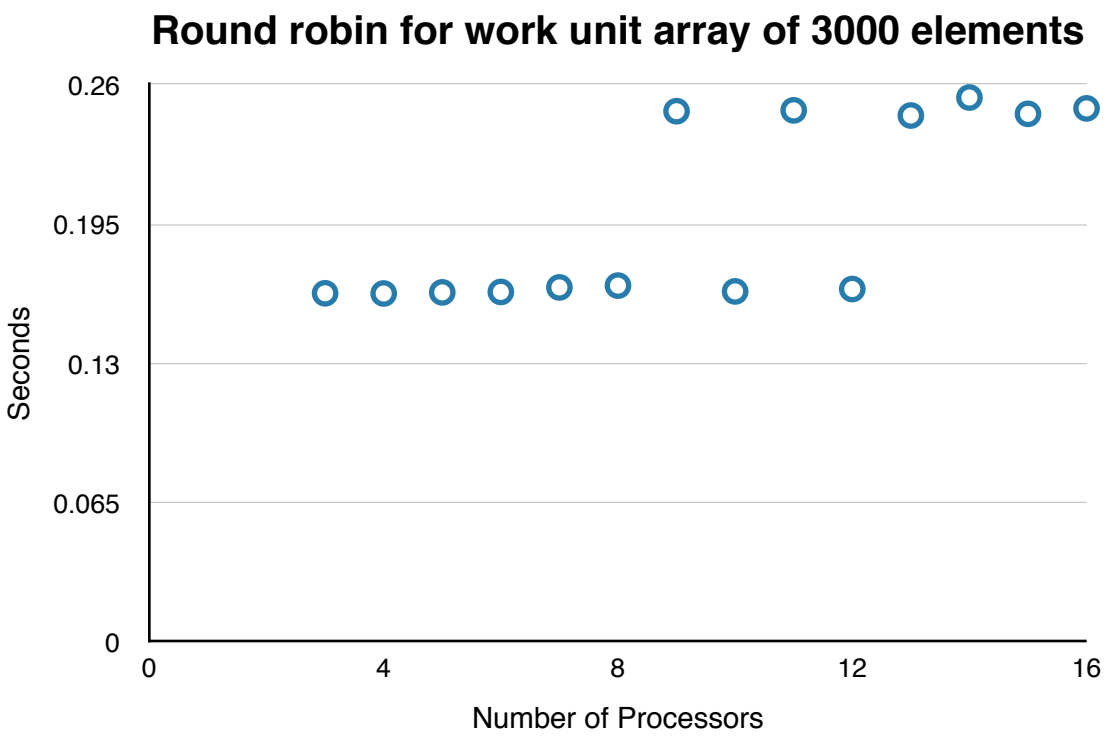
- `mpicc -lgmp api.c factor.c -o factor`
- `mpirun -np "number of processors" factor`

## Speedup Analysis

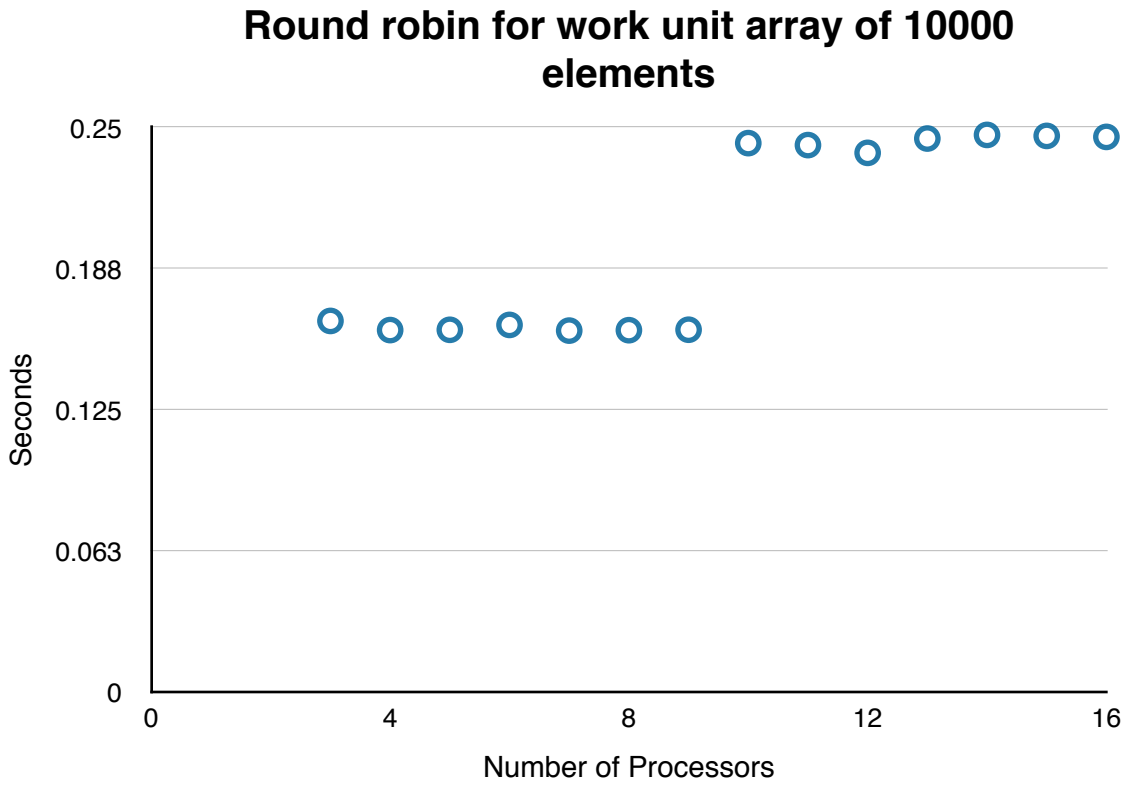
We now plot the speedup for our solution changing the number of processors. It is worth mentioning that we calculate granularity based on number of computations (square root of the large number) divided by the number of messages sent (which can be specified in factor.c by changing the *ARRAY_SIZE* macro). First we analyze the speedup for the **round-robin** technique (pre-assigning work to each worker) calculating the unique prime factors of 2626838234521:




**Granularity:** 1620753 computations / 3245 messages sent = 500 computations / message

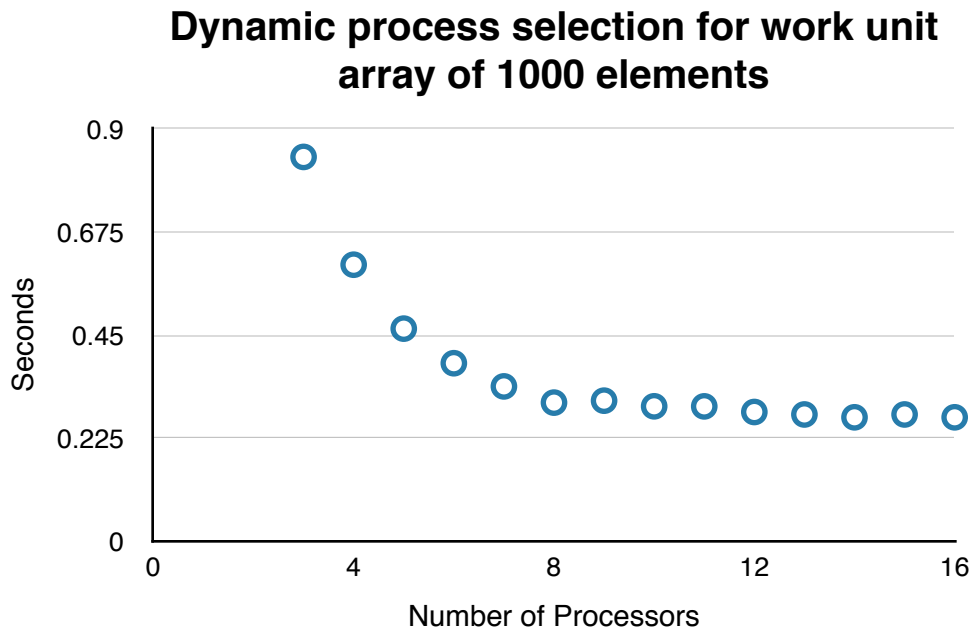



**Granularity**: 1620753 computations / 1095 messages sent = 1480 computations per message

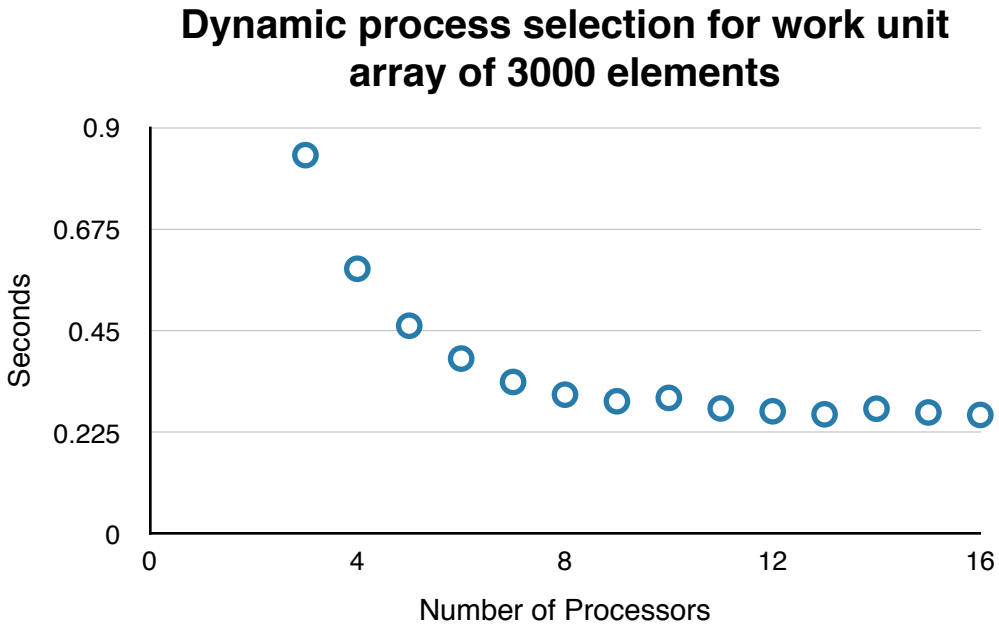



**Granularity**: 1620753 computations / 340 messages sent = 4766 computations per message

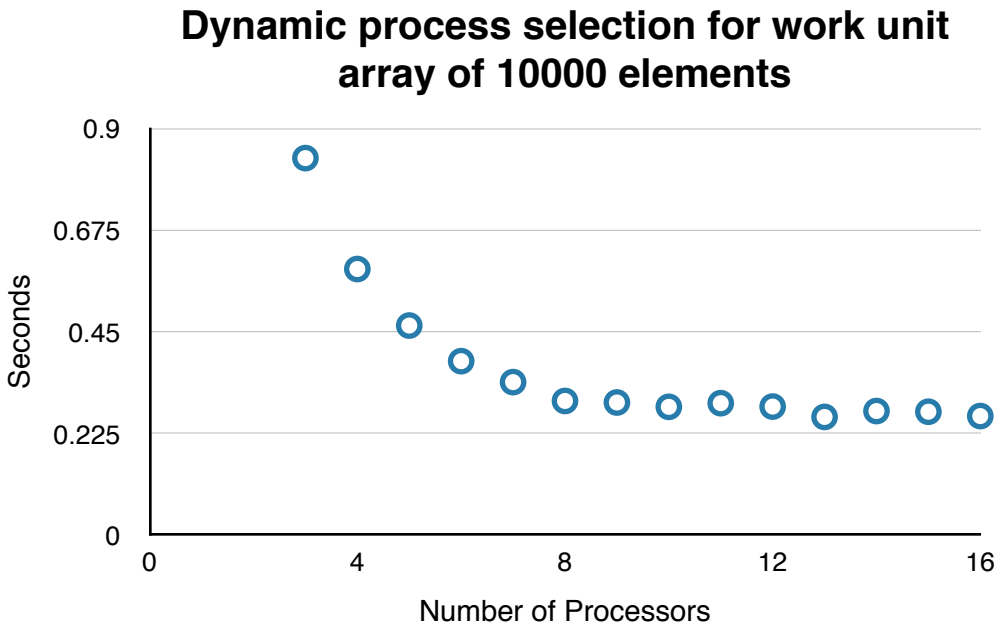
Now we analyze the speedup for the **dynamic-processor selection** technique calculating the unique prime factors of an even larger number, 262683328234521:




**Granularity**: 16207509 computations / 32420 messages sent = 500 computations per message




**Granularity:** 16207509 computations / 10820 messages sent = 1500 computations per message




**Granularity:** 16207509 computations / 3256 messages sent = 5000 computations per message

## Conclusion

From what we can see in our results, granularity is not particularly impactful in the performance of our parallel programs. If anything, though inconclusive, using the round-robin technique it seems that the algorithm is more stable for higher number of processors when the granularity is lower. It would however make sense to think that the more amount of computations per message help improve performance since less messages need to be sent, while the computations are still executed by as many processors as available.

In terms of comparing the round-robin technique against the dynamic processors allocation, it is clear that **assigning a process work dynamically has a significant advantage**.

*Note: it is important to mention that these results are from a EC2 c3.4xlarge machine. The c2.x4large machine was not available from AWS at the time of executing the algorithms.