A Parallel Algorithm for Enumerating Combinations

Martha Torres Alfredo Goldman
Junior Barrera
Departamento de Ciência da Computação
Instituto de Matemática e Estatística - Universidade de São Paulo
Rua do Matão, 1010 05508-900 São Paulo, Brazil
{mxtd, gold, jb} @ime.usp.br

Abstract

In this paper we propose an efficient parallel algorithm with simple static and dynamic scheduling for generating combinations. It can use any number of processors $(NP \leq n - m + 1)$ *in order to generate the set of all combinations of* $C(n, m)$. *The main characteristic of this algorithm is to require no integer larger than* n *during the whole computation. The performance results show that even without a perfect load balance, this algorithm has very good performance, mainly when* n *is large. Besides, the dynamic algorithm presents a good performance on heterogeneous parallel platforms.*[1]

1. Introduction

The enumeration of combinatorial objects occupies an important place in computer science due to its many applications in science and engineering [10][11]. Our special motivation for this topic is in genetic applications, in those applications, the generation of all combinations of m out of n objects (where m and n are genes) is necessary to analyze the interaction of genes in distinct conditions [6]. Due to the combinatorial behavior of this problem it is highly appropriated to develop parallel algorithms for it. In fact, there are many parallel solutions to generate the set of combinatorial objects (e.g., those in [1] [2][8][14]). These parallel algorithms can be divided in two classes. The algorithms which require a constant number of processors: [2][8][14], and the adaptive algorithms that use an arbitrary number of independent processors [1]. Usually, it is reasonable to assume that the number of processors on a parallel computer is not only fixed but also smaller than the size of a typical problem, in order to take advantage of the total capacity of platform.

The best adaptive algorithm is described in [1]. This algorithm can use any number of processors $(NP \leq C(n, m))$ and is optimal when uses k processors, where $1 \leq k \leq \frac{C(n,m)}{n}$. However, it requires arbitrary-precision arithmetic, moreover it is necessary to schedule the combinations, that is, to decide when each combination will be computed, before the moment where each processor can independently generate its combinations subset.

The aim of this paper is to present an efficient and simple parallel algorithm, using static and dynamic scheduling, in order to generate all combinations of m out of n objects in a distributed memory parallel machine using the message passing paradigm. This algorithm does not present the limitations of the previous algorithm [1]. In order to evaluate the performance of the proposed algorithm, we compared it with the algorithm of [1], and also with our algorithm using *largest-processing-time* (LPT) scheduling [3]. Those algorithms were implemented in C language and MPI library [12].

The paper is organized as follows. Section 2 describes the previous adaptive algorithm [1]. Section 3 explains the proposed algorithm using static scheduling. Section 4 shows our algorithm using dynamic scheduling. Section 5 presents real tests and a performance evaluation. Some concluding remarks are given in Section 6.

2. Classical Algorithm

Before the algorithm description, we present some definitions: an *m*-combination of an *n*-set is a subset with *m* elements chosen from a set with n elements. The number of *m*-combinations of an *n*-set is the binomial coefficient $C(n, m) = \frac{n!}{(m!(n-m)!)}$. The parallel algorithm in [1] uses an arbitrary number of independent processors (NP: number of processors $\leq C(n, m)$). Each processor generates a continuous interval of $\frac{C(n,m)}{NP}$ combinations. In this algo-

[1]This research was supported by Brazilian FAPESP, process number 00/10660-8 and CAPES.

rithm, each combination is associated with a unique integer. By using those integers, a processor can easily determine the first combination in the interval. After the first combination is generated, the remaining combinations in that interval can be easily obtained. The NP processors once started execute independently the same algorithm and do not communicate. Therefore, this algorithm requires:

1. to know the total number of combinations of m out of n objects $(C(n,m))$;
2. to have a numbering system for all combinations of m out of n objects. More specifically to have a function that provides a *k*-combination for a distinct integer. There are two functions called RANKC and $RANKC^{-1}$ [7] that realizes the combinatorial numbering system. Let $x = x_1x_2 \dots x_m$ be a combination of m integers. RANKC is a function which associates with each such combination x a unique integer $RANKC(X)$. The function RANKC has the following properties: (i) it preserves lexicographic ordering; (ii) its range is the set $1, 2, \dots, C(n,m)$; (iii) it is invertible such that if d = RANKC(X) then X can be obtained from $RANKC^{-1}(d)$ [1].
3. to have an algorithm that sequentially generates all combinations of m out of n objects in lexicographic order.

In this paper, the value C(*n*,*m*), in the item 1, is computed in O(*m*) time using the Algorithm 160 [13]. Algorithms for ranking and unranking combinations are widely used in solving combinatorial problems in parallel because they can be applied for parallel generation of combinatorial objects and also play an important role in the division of splitable tasks and on the distribution of resulting subtasks among cooperating processors [8].

Basically, unranking combinations algorithms can be characterized by the method applied for binomial coefficient evaluation. In the original formulation, the binomial coefficients were there derived by factorialing, in modern algorithms the next consecutive binomial coefficient is obtained by certain modifications of the previous one, it is called ”restricted factorialing”. Other approaches were proposed which binomial coefficients are picked up from a supplementary table. In this paper, we use the algorithm called UNRANKCOMB-D [8], which belongs to the ”restricted factorialing” category. This algorithm is very efficient in space and time. It presents only 0(*n*) time complexity.

The sequential combinations algorithm used in this paper is based in the Algorithm 154 [9] whose running time is O(*m*C(*n*,*m*)), which corresponds to an optimal algorithm [2].

The classical algorithm can use any number of processors (*NP* $\leq$ (C(*n*,*m*)) and is optimal when uses *k* processors, where $1 \leq k \leq$ C(*n*,*m*)/*n* [1]. But, it requires arbitrary-precision arithmetic because it has to deal with large integers during the computation of the total number of combinations and in the execution of unranking algorithm.

Moreover, for this algorithm, it is necessary to execute the routines for calculating C(*n*,*m*) and RANK^{-1} before each processor can independently generate its interval of consecutive combinations.

3 The Algorithm with Static Scheduling

First we will divide all combinations in groups, whose characteristic is the value of their “prefix”. For instance, the combinations of C(*5*,*3*) are: *012, 013, 014, 023, 024, 034, 123, 124, 134* and *234*. These combinations can be divided in three groups: the “group0” which is composed by the combinations whose “prefix” is *0* (*012, 013, 014, 023, 034*), the “group1” which is constituted by the combinations whose ”prefix” is *1* (*123, 124, 134*) and the ”group2” which is formed with the combinations whose ”prefix” is *2* (*234*). The total number of groups of all combinations of C(*n*,*m*) is *n-m+1*.

The proposed algorithm main idea is to divide all combinations in groups and to attribute the generation of combinations on each of these groups to the processors. In order to balance load we choose a static scheduling algorithm called reflexive wrap allocation. The distribution of groups is directly done without additional calculations.

Our solution using reflexive wrap allocation consists in: First, it attributes the correspondent group to the *myid* of each processor. Example: If we have a machine with two processors, the *myid* of processor *1* is *0* and the *myid* of processor *2* is *1*. Therefore, the processor *1* (*myid=0*) will generate the combinations of the “group0” and the processor *2* (*myid = 1*) those of the “group1”.

It is important to point out the number of combinations of the “group*i*” is larger than the number of combinations of the “group*i+1*”. Therefore, due to initial distribution, the processors with the lower *myid* will have to generate combinations of larger prefix group. In order to compensate the load imbalance, then in the following distribution, the subsequent groups with smaller “prefixes” will be assigned to processors with the larger *myid* and in the following distribution, the subsequent groups with smaller ”prefixes” will be assigned to processors with the smaller *myid* and so forth. Each processor stops generating combinations when the following group is larger than *n-m+1*

Therefore, the first attribution is made by the *myid* of each processor, the following designation (called “odd attribution”) depends on the total number of processors (*NP*), the *myid* of each processor and number of the previous group (*pg*). Thus, the corresponding group (*g*) for each processor is:

$$g = pg + 2 * (NP - myid - 1) + 1 \quad (1)$$

The following designation (called "even attribution") depends on the *myid* of each processor and the number of the previous group (*pg*). Thus, the corresponding group (*g*) for each processor is:

$$g = pg + 2 * myid + 1 \quad (2)$$

The next attribution acts according to the "odd attribution" relationship and the next succeeds the "even attribution" relationship, this repeats successively until the last group. The Figure 1 shows the size of tasks (groups) in this algorithm.

```
Group0=C(n-1,m-1)=(n-1)/(n-m)C(n-2,m-1)
Group1=C(n-2,m-1)=(n-2)/(n-m-1)C(n-3,m-1)
Group2=C(n-3,m-1)=(n-3)/(n-m-2)C(n-4,m-1)
.
.
.
Group(n-m-1)=C(n-n+m,m-1)=(m)C(n-1,m-1)
Group(n-m)=C(n-1,m-1)
```

Figure 1. The size of the tasks in the static algorithm

In our solution, the sequential algorithm used in each processor is the same used in the classical algorithm. In our solution, each processor knows which are the groups that belong to them, it does not generate any communication among the processors and it does not need the routines for calculating C(*n*,*m*) and RANK^{-1}.

3.1. The Algorithm with LPT Scheduling

In order to compare the performance of our algorithm, we implemented the LPT scheduling algorithm. In our solution, the size of each task (group) is known besides all tasks are currently independent of each other. Therefore, it can use the simple LPT scheduling algorithm. This method schedules the tasks (groups) one by one in decreasing order of processing time and each task is scheduled on the processor on which it finishes earliest. If t_{LPT} denotes the time for LPT schedule, t_{opt} the optimal time and *m* the number of processors available, then

$$t_{LPT} \leq \frac{4}{3} - \frac{1}{3m} \quad (3)$$

Thus, the generated schedule will never be worse than 4/3 of the optimal one. This type of scheduling provides a good performance but requires arbitrary-precision arithmetic in order to calculate the size of tasks as shown in the Figure 1.

It is obvious there are load balancing schemes better than LPT, however, as shown later, for our examples, the LPT algorithm presents a behavior very close to the optimal schedule. Therefore, it represents a good algorithm for performance comparison.

4. The Algorithm with Dynamic Scheduling

Though, the scheduling mechanism of static algorithm tries to distribute the combinations in balanced form, the load balance is not perfect. As the processors number increases, the imbalance of computational load increases, because the processors with *myid* bigger has to generate a smaller number of combinations. In order to improve the load balance, we propose an algorithm that uses dynamic scheduling. In this algorithm, a process is exclusively dedicated of distributing the groups for others processes. This process is called "master process". The distribution is done on demand:

1. Initially each process, with the exception of "master process", generates the combinations corresponding to the nominated group with its *myid*. For instance, the process with *myid=0* will generate the combinations of the "group0" and so forth for the other processes;

2. As soon as the processes generate the corresponding combinations, they will send a message to the master process, requesting the name of the group of the following combinations to generate. For this to be possible, the process sends its *myid* and wait that the master process sends to it, the name of the group;

3. The master process sends the groups to the processes, in the sequence the messages are received. This process sends a flag to each one of the other processes, when there are not more groups to send.

Observations:

The "master process" almost does not demand CPU, therefore this process can be executed together with other process in the same processor, without efficiency loss.

Performance results show that the use of two processes in one machine affects neither the execution time of the process which generates the combinations, nor the distribution of groups by the "master process". The dynamic algorithm can be used in heterogeneous clusters. To do so, for the first allocation we have to order the nodes of the cluster in non-decreasing order of speed.

5 Performance Results

We implemented the algorithms in a Beowulf-style cluster of 11 PCs. Each machine has a 1.2 GHz AMD Athlon K-7 processor, 768 MB of RAM, 2 MB CPU cache, and 30 GB hard disk space. The machines are interconnected with a 100 Mbps FastEthernet switch. The operating system is Linux 2.4.20 and we use C language and MPICH 1.2.4 library. In order to compare the performance of our proposed algorithm using static and dynamic scheduling, we measured the execution time by MPI_ Wtime for enumerating C(*100,6*), C(*50,10*), C(*100,8*), C(*1000,3*) and C(*2000,3*) combinations, using from two to eleven processors. The GNU MP library [4] (a very efficient library for arbitrary precision arithmetic on integers) was used on the LPT and classical algorithms. Figures 2-5 show the normalized total execution time (the total execution time of static algorithm is 1). The Figure 2 shows the comparison of total execution time for enumerating of C(*1000,3*) combinations. This Figure shows two different versions of the classical algorithm: **classical_ gmp** which uses the GNU MP library and **classical_ lint** that does not use it.

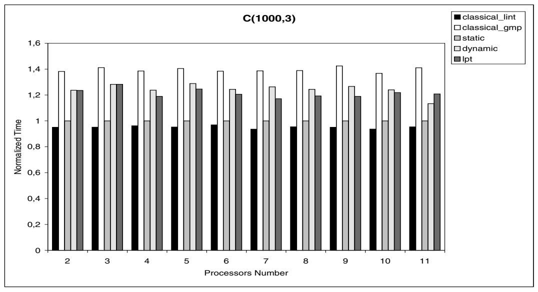

Figure 2. Comparison of total execution for enumerating C(*1000*,*3*) combinations using static, classical, LPT and dynamical algorithms

This figure exhibits **classical_ lint** is more efficient than other algorithms. The **static** algorithm presents an overhead of 5% for 11 processors.

The **classical_ gmp** algorithm presents an overhead of approximately 40% . And the dynamic and LPT algorithms present an overhead of approximately 20% (the LPT algorithm use the GNU MP library only when required).

It is important to stress that the **dynamic** algorithm is sensible to the tasks size. In this specific case, this algorithm will present contention because the task size in each processor is small (fine grain). Based in the Figure 1, the size of “group0” is C(*999,2*), the size of “group1” is C(*998,2*) and so on. For this reason, the master process must provide groups to the processes almost simultaneously. Using more than 3 processes, the demand for the scheduler process will increase and its contention will affect the performance. Therefore, we improve the **dynamic** algorithm for these situations increasing the task size for each process. In this case, the master process sends out tasks with *100* groups at once for each process.

The Figure 3 shows the comparison of total execution time for enumerating of C(*2000,3*) combinations. In this case as in the following examples, the **classical_ int** algorithm is not present in the figures anymore, the generated numbers can not be stored on long its. As shown in the Figure 3, the **static** algorithm is more efficient than other algorithms. This reduces the total execution time in approximately 30% compared with the dynamic algorithm, and in approximately 20% compared with the LPT algorithm.

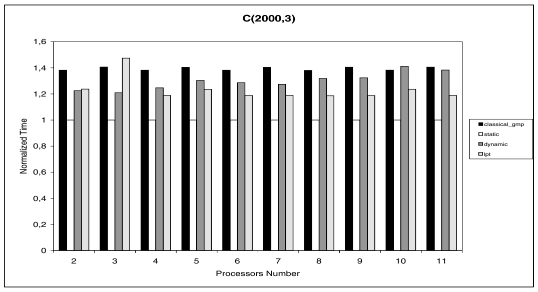

Figure 3. Comparison of total execution for enumerating C(*2000*,*3*) combinations using static, classical, LPT and dynamical algorithms

The dynamic algorithm presents a good performance but as the number of processor increases, its execution time increases, since it is affected by the overhead in communication operations.

In this case due to the size of the numbers, an arbitrary precision library must be used. The GMP MP library affects total execution time of the classical algorithm.

The Figure 4 shows the comparison of total execution time for enumerating of C(*100,6*) combinations. In this example, the static algorithm is more efficient than other algorithms. Moreover, the LPT algorithm presents a good performance mainly because it produces a good load distribution and does not have communication overhead.

In this case, the behavior of dynamic algorithm is also good because the task size for C(*100,6*) combinations is large; based in the Figure 1, the size of “group0” is C(*99,5*), the size of “group1” is C(*98,5*) and so on. Therefore the scheduler process does its work without contention. Moreover the overhead of communication operations almost does not affect the total execution time. As the processors number increases, the difference in total execution time for all

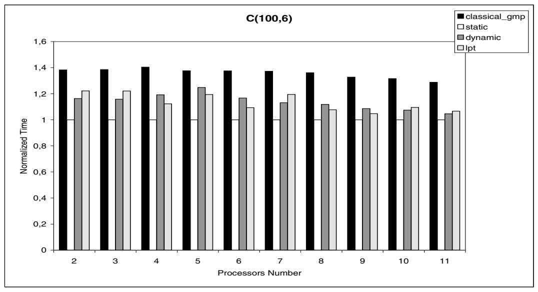

Figure 4. Comparison of total execution for enumerating C(*100*,*6*) combinations using static, classical, LPT and dynamical algorithms

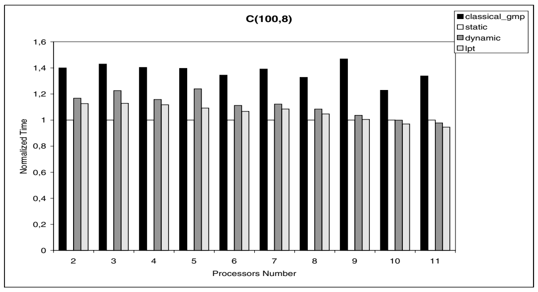

Figure 5. Comparison of total execution for enumerating C(*100*,*8*) combinations using static, classical and dynamical algorithms

algorithms is reduced compared to the static algorithm. The reason is the load balance provided by the static, it becomes imbalanced as exemplified latter in the paper.

The Figure 5 shows the comparison of total execution time for enumerating of C(*100,8*) combinations. In this case, the static, dynamic and LPT algorithms reveal a similar behavior.

The scheduling mechanism of static algorithm tries to distribute the combinations in balanced form, but the load balance is not perfect. As the processors number increases, the imbalance of computational load becomes larger, because the processors with larger *myid* have to generate a smaller number of combinations.

In this case, the LPT algorithm is efficient mainly because it produces a good load distribution. For example, the load balancing of LPT algorithm using $NP = 11$ processors exhibits a difference of 0.005% compared to the optimal schedule. For this situation, the static algorithm has an overhead of approximately 6% and the dynamic algorithm of approximately 3% .

The dynamic algorithm shows a good load balancing because tasks are course grain and the communication overhead does not affect the performance. Therefore, in situations like this where the static algorithm does not provide a suitable load balancing, the dynamic algorithm can be an option in order to provide a simple and efficient solution.

In order to illustrate the load balancing of static algorithm, the Figures 6 - 9 show the distribution of groups (tasks) for each processor, for numerating C(*100,8*) combinations in the static algorithm, using $NP = 5$ and 9 processors, respectively. The y axis shows the size of tasks (groups).

The Figure 6 shows the load balance is not perfect but it is better than the load balance presented in the Figure 8. Therefore, as the processors number increases, the imbalance of computational load also increases and so the total execution time of the static algorithm.

For comparison, the Figures 7 and 9 show the load balancing of LPT algorithm for the same cases. As shown, the LPT algorithm presents an excellent load balancing.

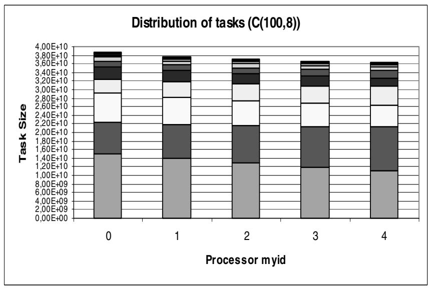

Figure 6. Distribution of tasks by static algorithm for enumerating C(*100*,*8*) combinations using NP = 5 processors

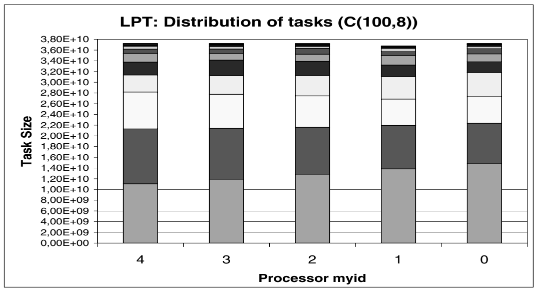

Figure 7. Distribution of tasks by LPT algorithm for enumerating C(*100*,*8*) combinations using NP = 5 processors

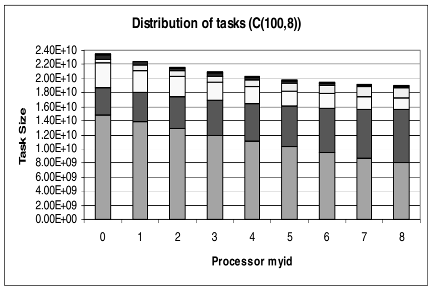

Figure 8. Distribution of tasks by static algorithm for enumerating C(*100*,*8*) combinations using *NP* = 9 processors

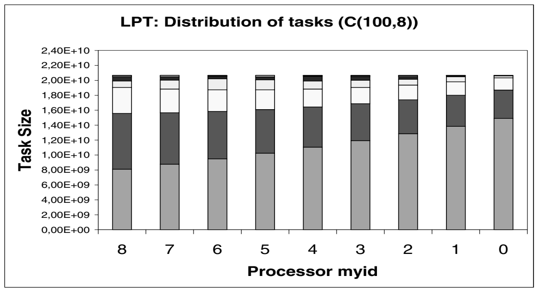

Figure 9. Distribution of tasks by LPT algorithm for enumerating C(*100*,*8*) combinations using *NP* = 9 processors

5.1. Special Case

The Figure 10 shows the comparison of total execution time for enumerating of C(*50,10*) combinations. In this example, the total execution time is normalized for classical algorithm.

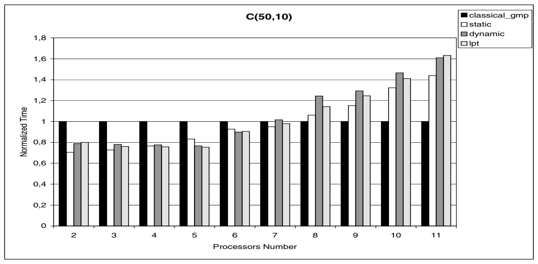

Figure 10. Comparison of total execution time for enumerating C(*50*,*10*) combinations using static, classical and dynamical algorithms

In this case, the classical algorithm presents the best behavior for $NP \geq 8$ processors. The figure 11 shows the distribution of groups (tasks) for each processor, for numerating C(*50,10*) combinations in the static algorithm, using *NP* = 5 processors.

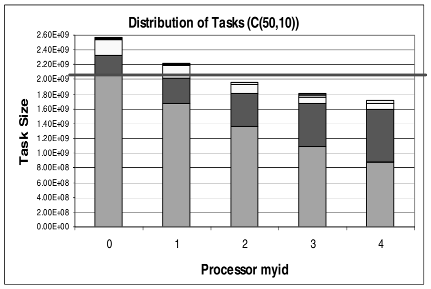

Figure 11. Distribution of tasks by static algorithm for enumerating C(*50*,*10*) combinations using *NP* = 5 processor

As illustrated in the Figure 11, the static algorithm presents a load imbalance which reduces its performance. In this case, only a limited number of tasks (3) affects the load balance for each processor, because the size of first task is larger than the other tasks. Figure 12 shows the distribution of groups (tasks) for each processor, for numerating C(*50,10*) combinations in the static algorithm, using *NP* = 6 processors. In this case, the load balance is even worse.

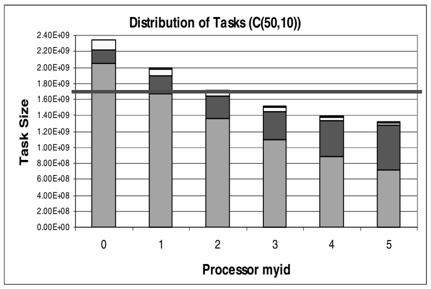

Figure 12. Distribution of tasks by static algorithm for enumerating C(*50*,*10*) combinations using *NP* = 6 processor

In the Figures 11 and 12, the horizontal line indicates the total combinations number divided by number of processors (*NP* = 5,6 respectively). The static algorithm provides a good performance when the "group0" size is minor or equal to total combinations number divided by the number of processors used. In this situation, the additional tasks on processor with *myid=0* can affect in a small degree the total execution time. If the "group0" size is larger than total combinations number divided by the number of processors used, it means the processor with *myid=0* will have more load and therefore its execution time is larger originating a load imbalance affecting the total performance. The number of combinations corresponding to "group0" is:

$$Comb = C(n-1, m-1) = \frac{m}{n} x C(n,m) \tag{4}$$

Therefore,

$$(\frac{m}{n})(C(n,m) \leq \frac{C(n,m)}{NP} \tag{5}$$

Then,

$$NP \leq \frac{n}{m} \tag{6}$$

In the case of dynamical algorithm, the minimal execution time that can be obtained is the execution time of "group0" by the processor with *myid*=0. Again, if the number of combinations corresponding to "group0" is smaller or equal to the total combinations number divided by number of processors. Then, as the number of processors increases, the total execution time decreases.

On the opposite situation, as the number of processes increases, the total execution time remains the same as the execution time of "group0" by the processor with *myid* =0. In this case, the classical algorithm can provide better performance.

In the example illustrated in the Figure 10 , the dynamical algorithm presents better performance until 7 processes, though the total execution time of dynamic algorithm since 5 processes is the same, the overhead of classical algorithm is only overcome using *NP* = 7 processors.

In the dynamic algorithm, the limit of performance is the size of first task, in other words, the minimum total execution time in the dynamic algorithm is the execution time of the first task.

The LPT algorithm presents similar behavior than dynamical algorithm therefore the same analysis may be applied to it.

Based in the Figure 1, the size of the last tasks (groups) is negligible compared to the first tasks. Therefore, if *n* is a large number, the size of tasks will be approximatelly equal, therefore the load balance improves for the static algorithm. If *m* is very small and *n* large, the size of tasks will also be very close.

5.2. Heterogeneous Behavior

In order to show the behavior of these algorithms in a heterogeneous platform, we simulated heterogeneous behavior loading five of eleven nodes with a program running in background while the execution of the algorithms was done. The Figure 13 shows the performance results.

The dynamical algorithm presents the best performance because it takes advantage of the nodes with larger capacity doing a good load balancing.

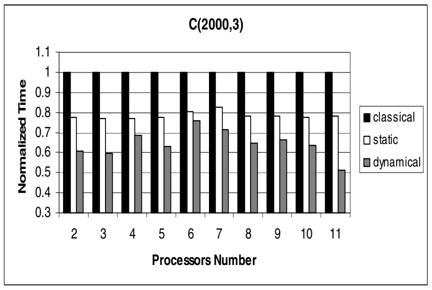

Figure 13. Comparison of total execution time for enumerating C(*2000*,*3*) combinations using static, classical and dynamical algorithms in a heterogeneous platform.

6. Conclusions

We presented a parallel algorithm for enumerating combinations with very basic static and dynamic scheduling algorithms that requires no integer larger than *n* during the computation. Our algorithm is simpler than the classical algorithm since it does not need to calculate the total number of combinations (C(*n*,*m*)) neither to have a numbering system for all combinations of *m* out of *n* objects (RANK^{-1}).

Our algorithm can use any number of processors ($NP \leq n-m+1$) in order to generate C(*n*,*m*) combinations.

The performance results show that our algorithm in its static version provides an efficient solution when NP $\leq n/m$. It produces an excellent performance when *n* is large. Our dynamic version is also very efficient especially with course grained tasks.

Besides, a combination of static and dynamic algorithms can be used in order to compensate the disadvantages of each other. For example in situations where the static algorithm does not provide a suitable load balancing, the dynamic algorithm can be used and in situations where the communication overhead of dynamic algorithm is affected then the static algorithm can be used.

Therefore, we considered our solution a good option because is simple and efficient.

The implementation of our algorithm using more elaborated scheduling mechanisms as LPT or bin packing [5] will provide a better performance especially when the number of tasks is relatively small and the number of processors is larger.

Also, the performance results show that the classical algorithm is affected by the overhead of special library for manipulating large integers.

The dynamic algorithm is also suitable for heterogeneous platforms.

References

[1] S.G. Akl. Adaptive and optimal parallel algorithms for enumerations permutations and combinations. *The comp. J.*, 30:433-436, 1987.

[2] S.G. Akl, D. Gries, and I. Stojmenovic. An optimal parallel algorithm for generating combinations. *Information Processing Letters*, 33:135-139, 1989.

[3] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal Applied Mathematics*, 17:416-429, 1969.

[4] T. Grandlund. Gnu mp. http://swox.org/gmp/.

[5] D.S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[6] S. Kim and et. al. Identification of combination gene sets of glioma classifications. *Mol. Cancer. Therapeutics.*, 1:1229-1236, 2002.

[7] G.D. Knott. A numbering system for combinations. *Communications of the ACM*, 17:45-46, 1974.

[8] Z. Kokosinski. Algorithms for unranking combinations and their applications. In *International Conference Parallel and Distributed Computing and Systems*, pages 216-224, 1995.

[9] C.S. Misfud. Combination in lexicographic order (algorithm 154). *Communications of the ACM*, 6:103, 1963.

[10] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Academic Press, second edition edition, 1978.

[11] F. Ruskey and C.A. Savage. A gray code for the combinations of a multiset. *Eurepean Journal of Combinatorics*, 17:493-500, 1996.

[12] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[13] M.L. Wolfson and H.V. Wright. Combinatorial of m thins taken n at a time (algorithm 160). *Communications of the ACM*, 6:106, 1963.

[14] B.B. Zhou, R. Brent, X. Qu, and W.F. Liang. A novel parallel algorithm for enumerating combinations. In *International Conference on Parallel Processing*, volume 2, pages 70-73, 1996.