Hierarchical Scheduling for Moldable Tasks

Pierre-François Dutot

Laboratoire ID-IMAG
38330 Montbonnot St-Martin, France
Pierre-Francois.Dutot@imag.fr

Abstract. The model of *moldable task* (MT) was introduced some years ago and has been proven to be an efficient way for implementing parallel applications. It considers a target application at a larger level of granularity than in other models (typically corresponding to numerical routines) where the tasks can themselves be executed in parallel on any number of processors. Clusters of SMPs (symmetric Multi-Processors) are a cost effective alternative to parallel supercomputers. Such hierarchical clusters are parallel systems made from m identical SMPs composed each by k identical processors. These architectures are more and more popular, however designing efficient software that take full advantage of such systems remains difficult. This work describes approximation algorithms for scheduling a set of tree precedence constrained moldable tasks for the minimization of the parallel execution time, with a scheme which is first used for two multi-processors and several bi-processors and then extended to the general case of any number of multi-processors. The best known approximations of competitive ratios for trees in the homogeneous case is 2.62, and although the hierarchical problem is harder our results are close as we obtain a ratio of 3.41 for two multi-processors, 3.73 for several bi-processors and 5.61 for the general case of several SMPs with a large number of processors. To our knowledge, this is the first work on precedence constrained moldable tasks on hierarchical platforms.

1 Introduction

In recent years computer hardware became increasingly affordable. This trends led to a greater number of parallel computers. However, a fast interconnection network is still very expensive. A solution to this problem is to use several processors on each motherboard connected by the network. This introduces a large difference in the time needed for on-board communications and for communications between two different motherboards.

In the case of Parallel Tasks (PT), where a task has to be processed by a fixed number of processors, the execution time of a task cannot be easily predicted on such hierarchical architectures unless some very restrictive hypothesis are made such as tasks have to be executed on one board only, or all communications are considered as long communications. We consider in this paper the related Moldable Task (MT) model, where the execution time of a task depends on the number of processors used to compute the task. However, in a hierarchical system knowing the number of processors used is not enough to predict the execution time, as communications can be local or distant. In [1], we provided a new hypothesis to deal with this problem. This placement hypothesis is

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 302–311, 2005.

recalled in Section 2. With this additional rule, the MT model is well suited to hierarchical systems.

Scheduling precedence constrained MT tasks is a NP-hard problem [2], and therefore approximation algorithms were developed to provide efficient schedules in polynomial time. The first approximation algorithm for the homogeneous case has been introduced by Lepère et al [3] with a ratio of 2.62 for tree based precedence constraints and a ratio of 5.24 for general graphs. This scheme has been recently improved by Hu Zhang in his PhD thesis [4–6] (under supervision of Pr. Jansen) achieving a 4.73 approximation ratio. In this paper, we adapted this scheduling technique of Lepère et al. in the case of tree precedence constrained moldable tasks, as a first step towards scheduling general graphs. To obtain ratios for general graphs without the improvements designed by Hu Zhang, the results presented here can be simply multiplied by a factor of 2. The recent improvements were not taken into consideration here due to the length limitation.

In the next section, we will recall the definitions of the Moldable Task model and its adaptation to hierarchical platforms. We will then briefly recall the scheduling scheme used for the homogeneous case. This scheme (and improvements by Zhang) will then be adapted for the two extremal cases of scheduling on two multi-processors and scheduling for several bi-processors. Finally a general scheme for scheduling on several multi-processors is proposed in Section 6.

2 The Moldable Tasks Model on Hierarchical Platforms

In the MT model a processor can compute only one task at a time, and the number of processors allocated to a task is constant during its whole execution. The execution time of a task depends on the number of processors allotted to it.

We consider an instance composed of n moldable tasks $\{T_1, \ldots, T_n\}$ to be scheduled on a cluster of m identical SMPs composed each of k identical processors. The tasks are linked with precedence constraints, in the form of trees (each node has at most one predecessor). The execution time of the moldable task T_i when allotted to p processors will be denoted by $t_i(p)$. Its *computational area* (or *work*) is defined as usually as the time space product $W_i(p) = pt_i(p)$. For a given allocation, we call *critical path* the maximum sum of execution times over a chain of the graph, and *work* of the graph, the sum of all the work of the tasks. The total work $W = \sum W_i(1)$ divided by mk, and the critical path L_{max} are straightforward lower bounds of the optimal makespan.

Using more than one processor to compute a task will cost some penalty for managing the communications and synchronizations. According to the usual behavior of the execution of parallel programs, we assume that the tasks are *monotonic*. This means that allocating more processors to a task will decrease its execution time and increase its computational area.

There exists a difficulty inherent to hierarchical systems due to the fact that communications inside the same SMP are faster than between processors belonging to different SMPs. In this case, the number of processors allotted to a task does not give all the informations needed to determine the execution time of a task: a task will be scheduled faster using processors inside the same SMP than using processors of different SMPs. In order to avoid this problem, we introduce below a dominant rule:

Definition (Best placement rule). *For a given number of processors, we say that a task is in its best placement if the penalty with this number of processors is the lowest possible.*

This definition is not very useful in the sense where many placements may verify the *best placement* condition, and from the definition we cannot decide where it is best to schedule the task. However, we can usually make the assumption that a task which runs on less than k processors will be in its *best placement* if all the processors allotted to the task are into the same SMP.

For tasks allotted to more than k processors, we need an additional hypothesis which is the following:

Hypothesis (Minimal penalty). *We assume in the rest of the paper that a task T_i allotted to a_ik+b_i processors (with $a_i \in [0;m]$ and $b_i \in [0;k-1]$) is in its best placement if exactly a_i SMPs are dedicated to it during its execution and the remaining b_i processors are within the same SMP.*

This hypothesis is clearly verified for clusters of bi-processors, as it avoids the cases where a task is sharing more than one bi-processor with other tasks. For larger values of k, this placement minimizes the number of clusters used by a task for a given allocation, therefore it is probably not far from the optimal placement.

Remark that we do not ask the processors to be contiguous. For instance, Figure 1 represents two tasks verifying the *minimal penalty* hypothesis. The third one does not.

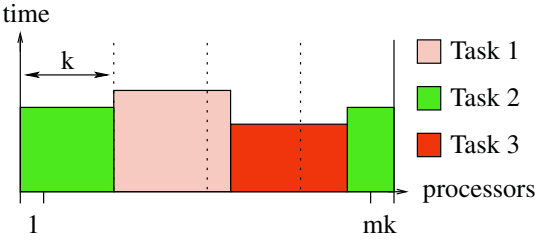

Fig. 1. Tasks 1 and 2 are in their *best placement*, whereas task 3 is not ($m = 4$).

In the rest of the paper, we will build algorithms whose output verify this best placement rule. However, the competitive ratios given are with respect to an optimal schedule which can use any kind of placement as long as the minimal penalty hypothesis holds, as the proof is based on the total workload.

3 Previous Results with Precedence Constraints

The schemes used in this paper are mainly inspired from the scheduling algorithm for the homogeneous case [3] (in this case $m = 1$). In this section, we will recall the basics of this algorithm.

In the homogeneous case, there is no placement problem ($k = 1$). The algorithm is composed of two phases. The first phase is a search for a good allocation for the

moldable tasks, i.e. an allocation which realizes a trade-off between the workload and the length of the critical path in the precedence graph. This problem is related to the general class of time-cost problems where the time needed to perform a task depends on the budget allotted to it. This problem has been solved by Skutella [7] very efficiently in the case of tree precedence constraints leading to an optimal trade-off, and also has good solutions for general graphs (leading to a 2 approximation on both the work and the critical path).

Once this allocation is known, all allocations greater than a parameter μ (i.e. all tasks using more than μ processors) are reduced to μ and then the second phase is a classic list scheduling algorithm. The analysis of the algorithm is similar to the classic proof of Graham's list scheduling algorithm, and for the best possible μ the performance ratio is $(3+\sqrt{5})/2 \simeq 2.62$ for trees and $3+\sqrt{5} \simeq 5.24$ for general graphs [3].

4 Scheduling with Two Multi-processors ($m = 2, k > 1$)

Schedules produced by the homogeneous algorithm are usually inadequate in a multi-processor setting, because of the placement rule. For a first view of the problem, we will consider in this section the restricted case of scheduling on two multi-processors.

To keep the same construction scheme as in the homogeneous case, we have to consider how the placement rule interferes in the list scheduling. As the parameter μ is less or equal to $mk/2$ in the homogeneous case, a task in its best placement cannot use processors in both multi-processors. We now distinguish two cases depending on the value of μ.

In the first case, for $\frac{2k+1}{3} < \mu \leq k$, the schedule produced by the list algorithm can be split into two kinds of time intervals. The first kind (of total length I_1) is composed of all the time intervals during which at most $2(k-\mu)+1$ processors are used. During these intervals, there are enough idle processors on at least one of the multi-processor to schedule a task. If those processors are idle there is no available tasks, which means that as in the original proof from Graham, a precedence constrained chain of tasks which covers all these intervals can be found. As $2(k-\mu)+1 < \mu$, the tasks in this chain did not have their allocation reduced to μ processors. The other kind of interval (of total length I_2) is composed of all the other time intervals. We denote by ω the length of the schedule.

With these two kinds of intervals defined, we can write the following (in)equalities:

$$\omega = I_1 + I_2 \tag{1}$$

$$\omega^* \geq L^*_{max} \geq I_1 \tag{2}$$

$$2k\omega^* \geq W^* \geq I_1 + 2(k-\mu+1)I_2 \tag{3}$$

where ω^* is the optimal makespan. The first one states that the total schedule length is the sum of all the time intervals, the second states that the critical path (and therefore the optimal schedule length) is greater than the length of the first kind of interval, and the third one is a lower bound on the workload in the optimal schedule.

A straightforward calculation proves that the ratio $\frac{\omega}{\omega^*}$ is at most equal to $\frac{4k-2\mu+1}{2(k-\mu+1)}$ which takes its minimum when μ is smallest, i.e. $\mu \leq \frac{2k+4}{3}$. The ratio is therefore bounded by $4 + \frac{3}{2(k-1)}$.

In the second case, for $\mu \leq \frac{2k+1}{3}$, the schedule can be split into three different kinds of time intervals. The first kind (of total length I_1) is when less than μ processors are used, the second kind (of length I_2) when between μ and $2(k-\mu)+1$ processors are used, and the third when at least $2(k-\mu+1)$ processors are used.

In the first and second kind of intervals, there is enough idle processors to schedule any tasks, therefore a chain of tasks covering all these intervals is again constructible. However this time, the tasks executed during intervals of the second kind may have been reduced from their original allocation to an allocation of size μ.

The previous (in)equalities are now:

$$\omega = I_1 + I_2 + I_3 \tag{4}$$

$$\omega^* \geq L^*_{max} \geq I_1 + \frac{\mu}{2k} I_2 \tag{5}$$

$$2k\omega^* \geq W^* \geq I_1 + \mu I_2 + 2(k-\mu+1) I_3 \tag{6}$$

To find the best upper bound for the performance ratio $\frac{\omega}{\omega^*}$, we can consider these inequalities as a set of linear programming constraints, where ω has to be maximized, and I_1, I_2 and I_3 are the variables. The dual problem is easier to solve, as there are only two variables. It is composed of the following (in)equalities:

$$z = \omega^* y_1 + 2k\omega^* y_2 \tag{7}$$

$$1 \leq y_1 + y_2 \tag{8}$$

$$1 \leq \frac{\mu}{2k} y_1 + \mu y_2 \tag{9}$$

$$1 \leq 2(k-\mu+1) y_2 \tag{10}$$

With the new objective of minimizing z. Combining equality 7 and inequality 9 we have $\frac{z}{\omega^*} \geq \frac{2k}{\mu}$, and adding $2(k-\mu+1)$ times inequality 8 to $2k-1$ time inequality 10, we get $\frac{z}{\omega^*} \geq 1 + \frac{2k-1}{2(k-\mu+1)}$. To minimize z we have to minimize the maximum of $\frac{2k}{\mu}$ and $1 + \frac{2k-1}{2(k-\mu+1)}$. The first quantity decreases when μ increases while the second quantity has the opposite behavior. The real minimum is therefore achieved when the two are equal, and the best μ is one of the two integers closest to the solution of $\frac{2k}{\mu} = 1 + \frac{2k-1}{2(k-\mu+1)}$, which is $\frac{8k+1-\sqrt{(8k+1)^2-32k(k+1)}}{4} \simeq (2-\sqrt{2})k + \frac{2+\sqrt{2}}{4\sqrt{2}}$. As k grows without bounds, this minimum gets close to $\frac{2}{2-\sqrt{2}} \simeq 3.41$. The value of the performance ratio for small values of k is given in Figure 2. With the exception of $k = 2$ where the ratio is 4, all the obtained performance ratio are less than $\frac{2}{2-\sqrt{2}}$, the minimum being 2.75 for k equal to four. Therefore it is always better to choose μ lower or equal to $(2k+1)/3$ for two multiprocessors.

Remark that if $\frac{2k}{\mu} \geq 1 + \frac{2k-1}{2(k-\mu+1)}$, the ratio is reached by a schedule of a single task. Let T_1 be a highly parallel task such as $t_1(p) = \frac{t_1(1)}{p}$, its optimal execution time would be $\frac{t_1(1)}{2k}$, and the schedule produced with our algorithm has an execution time of $\frac{t_1(1)}{\mu}$, leading to the ratio $\frac{2k}{\mu}$.

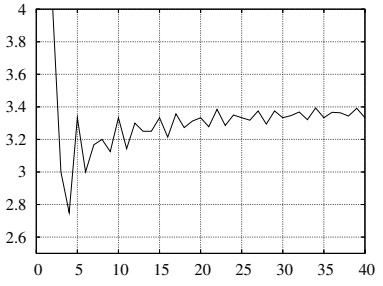

Fig. 2. Best performance ratio for two multi-processors of sizes up to 40 processors each.

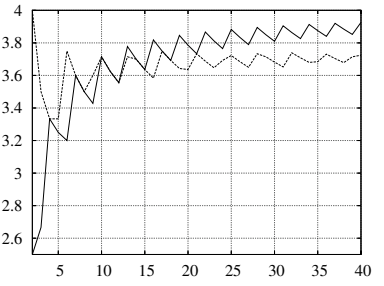

Fig. 3. Best performance ratio for up to 40 bi-processors. The dotted line is for $\mu \leq \frac{2m+1}{3}$, and the solid line for $\frac{2m+1}{3} < \mu$.

5 Scheduling on Bi-processors ($m \geq 2$, $k = 2$)

The second restricted case which is interesting to consider before addressing the general case, is scheduling on a large number of bi-processors. In this case, restricting the allocation to a portion of a bi-processor as we did previously makes no sense. The solution we considered is to directly use the homogeneous algorithm, with a different value for μ, and try to prove that the placement constraint with bi-processors is generally satisfiable.

Let m be the number of available bi-processors. As previously, we restrict the allocations of the first phase which are greater than μ to μ. The placement rule states that to place a task of allocation a, we need to have at least $\left\lfloor \frac{a}{2} \right\rfloor$ idle bi-processors plus eventually a processor if a is odd. As we did in the previous section, we will consider two cases depending on the value of μ.

For $\frac{2m+1}{3} < \mu \leq m$, the schedule can be split into two kinds of time intervals of respective length I_1 and I_2. The first kind of time intervals is when at most $m - \left\lfloor \frac{\mu}{2} \right\rfloor$ processors are used. In these intervals, there is enough idle processors to schedule a task using μ processors. All other time intervals are counted in the other kind of time interval.

As previously, we can write some inequalities on the length ω of the schedule produced by the algorithm:

$$\omega = I_1 + I_2 \tag{11}$$

$$\omega^* \geq L^*_{max} \geq I_1 \tag{12}$$

$$2m\omega^* \geq W^* \geq I_1 + \left(m - \left\lfloor \frac{\mu}{2} \right\rfloor + 1\right) I_2 \tag{13}$$

From these (in)equalities, it is straightforward to prove that:

$$\frac{\omega}{\omega^*} \leq \frac{3m - \left\lfloor \frac{\mu}{2} \right\rfloor}{m - \left\lfloor \frac{\mu}{2} \right\rfloor + 1} \tag{14}$$

which means that the best ratio is obtained for the smallest possible value of μ, which is $\left\lfloor \frac{2m+1}{3} \right\rfloor + 1$. This ratio is lower than 4 and tends to 4 for large values of m.

For smaller values of μ, i.e. $\mu \leq \frac{2m+1}{3}$, we again have to distinguish three kinds of time intervals, of respective length I_1, I_2 and I_3, depending on the number of processors used. The first kind is made of intervals where less then μ processors are used, the second kind is composed of intervals with a number of processors between μ and $m - \left\lfloor \frac{\mu}{2} \right\rfloor$ and the third of time intervals with more than $m - \left\lfloor \frac{\mu}{2} \right\rfloor$ busy processors.

Again, there is a set of (in)equalities describing the length of the schedule:

$$\omega = I_1 + I_2 + I_3 \tag{15}$$

$$\omega^* \geq L^*_{max} \geq I_1 + \frac{\mu}{2m} I_2 \tag{16}$$

$$2m\omega^* \geq W^* \geq I_1 + \mu I_2 + \left(m - \left\lfloor \frac{\mu}{2} \right\rfloor + 1\right) I_3 \tag{17}$$

Which can be seen as a linear programming set of equations, and the dual is this time:

$$z = \omega^* y_1 + 2m\omega^* y_2 \tag{18}$$

$$1 \leq y_1 + y_2 \tag{19}$$

$$1 \leq \frac{\mu}{2m} y_1 + \mu y_2 \tag{20}$$

$$1 \leq \left(m - \left\lfloor \frac{\mu}{2} \right\rfloor + 1\right) y_2 \tag{21}$$

As before, some straightforward rewriting yields to:

$$\frac{z}{\omega^*} \geq \frac{2m}{\mu} \tag{22}$$

$$\frac{z}{\omega^*} \geq 1 + \frac{2m-1}{m - \left\lfloor \frac{\mu}{2} \right\rfloor + 1} \tag{23}$$

Again, we have to find the μ which will minimize the maximum of the two lower bounds. This time, the best μ can be bounded between two functions of m:

$$\left\lceil 4m - 1 - \sqrt{12m^2 + 4m + 1} \right\rceil - 1 \leq \mu \tag{24}$$

$$\mu \leq \left\lfloor 4m - \sqrt{12m^2 - 8m} \right\rfloor + 1 \tag{25}$$

The obtained performance ratio is presented in Figure 3, with a dotted line for small values of μ and a solid line for large values of μ. When the number of bi-processors is lower than ten, the best solution is achieved with a large μ, whereas for more bi-processors, μ has to be smaller. As m grows without bounds, $\frac{\mu}{m}$ gets close to $(4 - 2\sqrt{3})$ and the performance ratio of the algorithm tends to $\frac{1}{2-\sqrt{3}} \simeq 3.73$.

6 A General Framework ($m > 2, k > 2$)

The algorithms of the two previous sections cannot easily be extended to an arbitrary number of multi-processors with a large number of processors. The number of multi-processors m is a lower bound on the ratio of the first algorithm, as μ is always lower than k, while k is a lower bound of the ratio of the second one as m sequential tasks

can prevent the execution of tasks allotted to at least k processors. A closer look shows that the first algorithm corresponds to $\mu < k$, and the second one to $\mu \geq k$.

To design efficient schedules for the general case, we have to take the best of the two previous algorithms, considering both the tasks with a large allocation and the tasks with a small allocation. The main idea is to use different values μ for small and large tasks, and then restrict the execution of the small tasks on a specific part of the platform.

For the rest of the paper, we consider m multi-processors, having k processors each. Let γ be an integer between 1 and m, γ sets the threshold between "small" and "large" tasks. Tasks allotted to less than γk processors are "small", while other tasks are "large". As we will need two different values of μ for small and large tasks, we will keep the μ notation for small tasks, and denote by δk the largest allotment allowed (hence δk plays the same role for large task as μ does for small tasks).

After the first allotment phase, the allotment of the tasks is reduced in the following way:

- Tasks allotted to a processors, with $a \leq \mu$ are kept in their original allotment.
- Tasks allotted to a processors, with $\mu < a < \gamma k$ are reduced to μ processors.
- Tasks allotted to a processors, with $\gamma k \leq a < \delta k$ are reduced to $\left\lfloor \frac{a}{k} \right\rfloor k$ processors.
- Tasks allotted to a processors, with $\delta k \leq a$ are reduced to δk processors.

Once this allotment is determined, the schedule is produced by a list scheduling algorithm, with always at most θ multi-processors[1] filled with small tasks. However, the large tasks can fill more than $(m-\theta)$ multi-processors if there is not enough small tasks. As previously, we can split the resulting schedule in several kind of time intervals, depending on occ_{small} and occ_{large} which are the number of processors used respectively by small and large tasks:

- S_1 is the set of intervals such as $1 \leq occ_{small} < \mu$ and $occ_{large} = 0$. In all the time intervals of this set, there is always a task which is part of the constructed critical path, and whose allocation has not been reduced.
- S_2 is the set of intervals such as $\mu \leq occ_{small} < \theta(k - \mu + 1)$ and $occ_{large} = 0$. In all the time intervals of this set, there is always a task which is part of the constructed critical path, and whose allocation may have been reduced to μ.
- S_3 is the set of intervals such as $\gamma k \leq occ_{large} < \delta k$ and $occ_{small} = 0$. In all the time intervals of this set, there is always a task which is part of the constructed critical path, and whose allocation has been reduced to the nearest multiple of k.
- S_4 is the set of intervals such as $\delta k \leq occ_{large} < (m - \delta + 1)k$ and $occ_{small} = 0$. In all the time intervals of this set, there is always a task which is part of the constructed critical path, and whose allocation may have been reduced to δk.
- $S_{critical}$ is the set of intervals which are not in the previous sets, and where you can still schedule a task, either small or large. Mathematically, the occupations are either $occ_{large} < (m - \theta - \delta + 1 + a)k$ and $occ_{small} \leq \theta - a$ for a between 1 and θ, or $occ_{large} < (m - \theta - \delta + 1)k$ and $occ_{small} < \theta(k - \mu + 1)$. We can redistribute all the time intervals from this set to sets S_1 to S_4, depending on the task of the interval which is considered for building the critical path.

[1] Please note that these θ SMPs are not fixed. If a small task is ready and less than θ SMPs are used by small tasks, any available SMP can be partially used by the small task.

- S_5 is the set of intervals such as $\theta(k-\mu+1) \leq occ_{small}$. In these time intervals, if a task of size μ is available, it may be impossible to schedule it.
- S_6 is the set of intervals such as $(m-\delta-\theta+1)k \leq occ_{large}$ and $m+1-\delta-\frac{occ_{large}}{k} \leq occ_{small}$. In these time intervals, if there is an available task of size δk, it may be impossible to schedule it.

Remark that some of these intervals may be empty, and some are overlapping. Depending on the values of θ, k and μ, S_2 can be empty. If this is the case, the upper bound on occ_{small} of S_1 is reduced to meet the upper bound of S_2. In the same way, depending on the values of m and δ, S_4 may be empty. Again, if this is the case, the upper bound of S_3 must be reduced to the upper bound of S_4. Time intervals which can be in S_5 and S_6 are put in the set S_5 if $\theta(k-\mu+1) > (m-\delta-\theta+1)k+\theta$ and in set S_6 otherwise.

As previously, denoting I_x the total length of the intervals in set S_x, we can bound the length of the intervals with the total workload and the critical path:

$$\omega = I_1 + I_2 + I_3 + I_4 + I_5 + I_6 \tag{26}$$

$$\omega^* \geq I_1 + \frac{\mu}{\gamma k - 1} I_2 + \frac{\gamma k}{(\gamma+1)k-1} I_3 + \frac{\delta}{m} I_4 \tag{27}$$

$$\begin{aligned} mk\omega^* \geq\ & I_1 + \mu I_2 + \gamma k I_3 + \delta k I_4 + \theta(k-\mu+1) I_5 \\ & + \left((m-\delta-\theta+1)k+\theta\right) I_6 \end{aligned} \tag{28}$$

And from these equations, we can write the dual problem:

$$z = \omega^* y_1 + mk\omega^* y_2 \tag{29}$$

$$1 \leq y_1 + y_2 \tag{30}$$

$$1 \leq \frac{\mu}{\gamma k - 1} y_1 + \mu y_2 \tag{31}$$

$$1 \leq \frac{\gamma k}{(\gamma+1)k-1} y_1 + \gamma k y_2 \tag{32}$$

$$1 \leq \frac{\delta}{m} y_1 + \delta k y_2 \tag{33}$$

$$1 \leq \theta(k-\mu+1) y_2 \tag{34}$$

$$1 \leq \left((m-\delta-\theta+1)k+\theta\right) y_2 \tag{35}$$

Although it may seem much more complicated, this problem is still two dimensional and the extremal point of the polytope can be found. Due to the restrictions on the paper length the case analysis will not be presented here, but is instead provided in an extended version of this paper [8]. Unsurprisingly the guarantees for the general case are not as good as in the two special cases studied in the previous sections. These results are summarized in Figure 4 and Figure 5.

We can see in these figures that the performance ratio is quickly worse than 4, and does not get bigger than 5.5 for small values of k and m. For very large values of k and m, this ratio tends to 5.61.

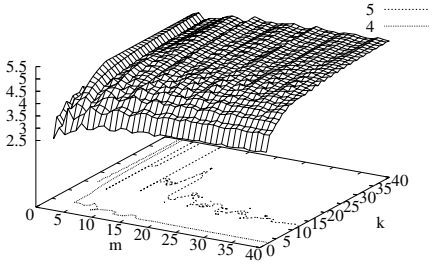

Fig. 4. Performance ratios for up to 40 SMPs having each up to 40 processors.

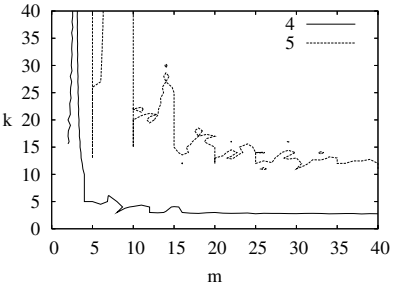

Fig. 5. Projections of the iso-levels 4 and 5 of Figure 4.

7 Conclusion

The algorithms presented in this paper are (to our knowledge) the first to address the problem of scheduling moldable tasks on hierarchical platforms. The next step is to add the improvements from Hu Zhang. In the longer run, we should implement the resulting algorithms in operational resource management systems. This implementation has to be preceded by a simulation phase, as the behavior of the algorithms on real workloads can be quite different from expected.

References

1. Dutot, P.F., Trystram, D.: Scheduling on hierarchical clusters using malleable tasks. In: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, ACM Press (2001) 199–208
2. Du, J., Leung, J.T.: Complexity of scheduling parallel tasks systems. SIAM Journal on Discrete Mathematics **2** (1989) 473–487
3. Lepere, R., Trystram, D., Woeginger, G.: Approximation algorithms for scheduling malleable tasks under precedence constraints. In Springer-Verlag, ed.: 9th Annual European Symposium on Algorithms - ESA 2001. Number 2161 in LNCS (2001) 146–157
4. Zhang, H.: Approximation Algorithms for Min-Max Resource Sharing and Malleable Tasks Scheduling. PhD thesis, University of Kiel, Germany (2004)
5. Jansen, K., Zhang, H.: Scheduling malleable tasks with precedence constraints. In: 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005), Las Vegas (2005)
6. Jansen, K., Zhang, H.: An approximation algorithm for scheduling malleable tasks under general precedence constraints (2005) submitted.
7. Skutella, M.: Approximation algorithms for the discrete time-cost tradeoff problem. Mathematics of Operations Research **23** (1998) 909–929
8. Dutot, P.F.: Hierarchical scheduling for moldable tasks – extended version. Technical report, Laboratory ID-IMAG (2005) `www-id.imag.fr/~pfdutot/perso.html`.