Optimal Scheduling Results for Parallel Computing and Other Contexts

Håkan Lennerstad
Dept. of Mathematics
Univ. of Karlskrona/Ronneby
S-371 79 Karlskrona
Hakan.Lennerstad@itk.hk-r.se

Lars Lundberg
Dept. of Computer Science
Univ. of Karlskrona/Ronneby
S-372 25 Ronneby
Lars.Lundberg@ide.hk-r.se

1 General Scheduling

The situation of having jobs of some kind which may be dependent, and having several possible executors of jobs, is a very common one and appears in very diverse contexts. Clearly some kind of scheduling is needed: who shall do what? Also one often wants to have it all done as quickly as possible. One central scheduling question is whether to allow transferring of jobs between executors. The antipoles here are allowing unlimited transferring, usually called dynamic allocation of jobs, and allowing no transferring, which is static allocation.

2 Parallel Program Scheduling

Perhaps the most conventionalized case of scheduling is the one occurring when executing a parallel program on a parallel computer. Here one often have identical processors, and the jobs themselves are usually not affected by the way they are scheduled. The results presented here concern such a case.

We describe combinatorial functions optimally comparing optimal static to optimal dynamic allocations, i.e. allocations of the two kinds which minimize the total completion time. The functions give the ratio of the execution times in the worst case: it is a maximum of the ratio over all parallel programs with any interprocess dependency structure, only exempting deadlock. The behaviour of this worst case ratio as a function of the number of processes and the number of processors is also described.

Immediate applications can be found within parallel computer design and evaluation of static allocation algorithms. The results have arisen in the context of parallel computing, and are here presented in this terminology.

3 Choices of Scheduling

Consider having a parallel computer with k identical processors and a parallel program P with n processes. In the usual, worst and thus most interesting case, the program P has a lot of dependence between its processes - at several time points some processes cannot execute unless some other processes have completed certain calculations. Suppose further that $n > k$, hence some processor has to execute at least two processes. If $n \leq k$ there is no scheduling problem, then each process can occupy a processor on its own.

An essential performance issue is how to distribute the processes over the processors in order to minimize the total execution time. A parallel program P can obviously be allocated to the processors in many different ways, with varying total execution time: we have a basic set consisting of all possible allocations. In some of these no process is transferred, this is the subset of all static allocations. An allocation where the total execution time is minimal within this subset represents an optimal static allocation, and the execution time for the program P with an optimal static allocation is denoted by $T_s(P)$. The execution time for the program P with an optimal dynamic allocation is analogously denoted by $T_d(P)$.

Static allocation is a more restricted scheduling scheme than dynamic. Clearly, the execution time for a parallel program P with optimal static allocation is thus never lower than with an optimal dynamic allocation. The transferring of processes allowed in dynamic allocation, however, is sometimes complicated and can be time consuming, suggesting that the "clearly" above should be replaced by something less definite. A question which really concerns parallel computer designers, and which is the subject of this article, is: How much worse can it actually be to apply static allocation compared to dynamic?

4 Np-hard Scheduling Problems

The problems of finding optimal static and dynamic allocations are both known to be np-hard. In spite of this one can thus easily compare the per-

formance of the two allocation schemes. The execution time of the present formulas increases slowly with n but faster with k . Experiments show that this computation can favourable be parallelized, allowing values for large n and k to be computed. It is not known whether the execution time increases exponentially with k .

Next we make two standard neglections. We neglect the execution time of dependency signals between processes, which usually is neglectible, and also the cost of tranferring processes, which often is not neglectible. We thus now turn to a situation where the "clearly" above is fully valid, and where, given an executable parallel program P , the execution times with optimal static and dynamic allocations, $T_s(P)$ and $T_d(P)$ respectively, are well defined.

In this notation we present an explicit formula for the function

$$g(n, k) = \max_P \frac{T_s(P)}{T_d(P)},$$

where the maximum is taken over all parallel programs P with n processes, executed on a multiprocessor with k processors.

5 The Process Dependent Function

If $n \geq k$, we define $g(n, k) = 1$. Otherwise, if $w = n/k$ is an integer:

$$g(n, k) = \frac{1}{\binom{n}{k}} \sum_{l=1}^{\min(w, k)} l \pi(k, w, k, l),$$

where $\pi(k, w, q, l)$ is defined below. If $w = n/k$ is not an integer, we let $w = \lfloor n/k \rfloor$ and denote the remainder of n divided by k by n_k , i.e. $n_k = n - k \lfloor n/k \rfloor$. Then

$$g(n, k) = \frac{1}{\binom{n}{k}} \sum_{l_1=\max(0, \lceil \frac{k-(k-n_k)w}{n_k} \rceil)}^{\min(w+1, k)} \sum_{l_2=\max(0, \lceil \frac{k-l_1 n_k}{k-n_k} \rceil)}^{\min(w, k-l_1)} \max(l_1, l_2) \times$$

$$\sum_{i=\max(l_1, k-l_2(k-n_k))}^{\min(l_1 n_k, k-l_2)} \pi(n_k, w+1, i, l_1) \pi(k-n_k, w, k-i, l_2).$$

The function $\pi(k, w, q, l)$ is 0 if $\min(q, w) < l$ or if $q > kl$, otherwise if $k = 1$, $\pi(k, w, q, l) = \binom{n}{q}$. In all other cases it is given by

$$\pi(k, w, q, l) = \binom{w}{l} \sum_I \binom{w}{i_1} \cdot \dots \cdot \binom{w}{i_{k-1}} \frac{k!}{\prod_{j=1}^{b(\{l\}+I)} a(\{l\} + I, j)!}.$$

Here the sum is taken over all sequences of nonnegative integers $I = \{i_1, \dots, i_{k-1}\}$ which are decreasing; $i_j \geq i_{j+1}$ for all $j = 1, \dots, k-2$, bounded by l ; $i_1 \leq l$, and have sum $q - l$; $\sum_{j=1}^{k-1} i_j = q - l$. By definition $b(I)$ is the number of distinct integers in I , and $a(I, j)$ is the number of occurrences of the j :th distinct integer in I . The notation $\{l\} + I = \{l, i_1, \dots, i_{k-1}\}$ is used.

The maximum of $g(n, k)$ for n and k up to 50 is 2.5430. It follows that optimal static allocation has for no parallel program of at most 50 processes, more than a factor 2.543 longer execution time than optimal dynamic allocation. The ratio itself is optimal: there are programs having this value. Parallel computer scientists have found this ratio surprisingly low.

The cost of transferring processes is probably strongly application dependent. In any case, taking this into account favours static allocation even more. On the other hand it is in practice easier to find close to optimal dynamic allocations than close to optimal static allocations. The formula gives quantitative information on how close, for the static case.

6 The Mathematical Formulation

We here very briefly and intuitively sketch the mathematical reformulation, for a full presentation see [2] or [4].

The formula can be deduced by reformulating the problem about parallel programs into a problem about 0,1-matrices. A 0,1-matrix represents a program P with optimal dynamic allocation. The matrix consists of n column vectors representing the processes, the rows represent time intervals of equal length. 1 denotes active and 0 denotes passive. For a given program, the time discretization is done so that each dependency signal and process activity change occur at an end point of a time interval. The number of intervals may thus be very large. Each row has k 1:s, one for each processor, since the worst case programs keep all processors constantly busy. A static allocation is represented by a partition of the set of column vectors. The execution time

with a specific static allocation is computed by adding the column vectors in each partition group together, resulting in k vectors representing the load on the k processors for each time interval. Next componentwise maximum is taken, since at each time interval the processor with maximal load have to be awaited. This results in a single remaining vector, the multiprocessor load for each interval. The sum of its entries is the execution time with this static allocation. In this formulation the execution time with optimal dynamic allocation is the number of rows, hence the ratio can be described as the arithmetic mean of the entries in the multiprocessor load vector. It is proved that a matrix where the rows are exactly all $\binom{n}{k}$ permutations of the k 1:s in n positions represent extremal programs, maximizing the ratio. Also there is always an optimal static allocation where the sizes of the sets in the partition differ as little as possible. When having reached this situation the formula for $g(n, k)$ can be derived.

7 Optimal Topography

The function $g(n, k)$ can be plotted as a function of two variables, see Figure 1. As a graph it has an interesting topography and perhaps surprising properties. Main parts of the plot can be described as plateaus and transition areas between them.

The plateau $g = 1$ is immediately visible, this is the trivial case when there are more processors than processes. Here no processes ever need to be transferred, there is no difference between static and dynamic allocations, hence the ratio is one.

The plateau $g = 2$ is also visible within $n, k \leq 50$. When moving from the first plateau, from the point $k = n = 15$, say, to $k = 15, n = 16$, one can observe in Figure 1 a sharp jump. In the 0,1-matrixformulation we here have a worst case matrix (parallel program) whose rows are the 16 permutations of 15 1:s and one single 0. In the partition (static allocation) two columns (processes) will belong to the same partition set (execute on the same processor). When adding the column vectors we will get 14 2:s and 2 1:s (in most time intervals this processor will have two busy processes), giving $g(16, 15) = 1 + 14/16$, which is almost 2.

The plateau $g = 3$ is not visible in the domain $n, k \leq 50$. It is established that for each positive integer w , $g(n, k)$ has a plateau $g = w$, i.e. $|g(n, k) - w|$ is arbitrary small for all points (n, k) in an unbounded domain sufficiently far away between the "straight lines" $n = (w - 1)k$ and

$n = wk$. The main unknown geometrical feature about the function $g(n, k)$ is perhaps the distance from the origin to each plateau.

8 Optimal Ridge Mountains

Valleys and ridges can be observed in the plot. Before the plateaus are fully developed they have the form of a "mountain ridge". As is revealed in Fig. 2 the plateau $g = 2$ actually has a ridge in the middle, although very flat. Observed from the " n -axis", and perpendicularly to it, it is the highest ridge until $n = 10$. The second ridge starts at $n = 10$, takes over as the highest at $n = 11$, until $n = 23$, where it is surpassed by the third ridge. At $n = 36$ the first sign of the fourth ridge appear. The last two ridges are strongly expected to eventually develop to the plateaus $g = 3$ and $g = 4$. The plots suggest that every following ridge raises to the plateau level very much slower than the previous one. Hence the transition areas between the plateaus similarly become larger.

9 Optimal Horizon Function

Even more definite things can be said when viewing the plot from the k -axis. The function $g(n, k)$ is increasing as a function of n - when the number of processes increases, parallel programs can be found with a larger ratio of optimal static versus dynamic allocations. This is an obvious truth for computer scientists, and can be proven within this mathematical framework. It turns out that it is also possible to calculate an explicit formula for the limit $\lim_{n \rightarrow \infty} g(n, k) = G(k)$. In the plot of $g(n, k)$, the function $G(k)$ represents the horizon when viewing the landscape of $g(n, k)$ from the k -axis, in the direction of the n -axis. In parallel computing terms the function $G(k)$ is the worst case ratio of optimal static and dynamic allocations for a parallel computer with k processors, where a worst case program is found in the domain of all parallel programs of any number of processes.

10 The Process Independent Function

With the same notation as above, here is the formula for the horizon function $G(k)$ ([3] and [5]):

$$G(k) = \frac{(k!)^2}{k^k} \sum_{l=1}^{l=k} \frac{1}{(l-1)!} \sum_I (\Pi_{j=1}^{k-1} i_j! \Pi_{j=1}^{b(\{l\}+I)} a(\{l\} + I, j)!)^{-1}.$$

The sum over I is taken over the same sequences as in the previous formula, with $q = k$.

Here we have a factor $(k!)^2 k^{-k}$ which tend to infinity rapidly, times a sum over certain decreasing sequences. The number of terms in this sum tend to infinity rapidly. On the other hand the terms of the sum are inverse values of products of factorials, i.e. numbers which are certainly very small and tend to zero very rapidly as k becomes large. These numerical aspects balance each other into a moderate behaviour of $G(k)$. $G(k)$ stays slightly larger than $\log_e k$ up to $k = 38$, but $G(39) = 3.65393 < 3.66356 = \log_e 39$.

11 Further Applications and Generalizations

This problem has appeared within parallel computing. However mathematical results have evolved, and the results could thus very well be applicable in other contexts. The construction of a house is one example. Here a lot of tasks need to be fulfilled, where some can be done only after other are completed, or partially completed. The processes here are the work tasks, the processors are the workers. Static allocation means that the worker who starts a task always completes it, while in dynamic allocation a task may be taken over by another worker. However, the hypothesis of identical processors is here usually not valid: humans tend to be specialists. This, as further parallel computer problems, calls for generalization in one direction: the case of non-identical processors.

Scheduling problems are very common within the area of telecommunications, hence this is a potential domain of application for enhancing performance.

12 Previous and Future Results

The only previous general results of similar type are the results by R. L. Graham [1]. In [3] it is established that $\lim_{k \rightarrow \infty} G(k) = \infty$, as expected. In [1] it

is proved that when considering only parallel programs with no interprocess dependency, this limit is 2. Another result of [1] concerns a dynamic allocation scheme called self-scheduling. Self-scheduling has the property that any time when a processor becomes idle and there are waiting executable processes, one of these is immediately allocated to this processor. In [1] it is also proved that the execution time with a self-scheduling algorithm is never longer than 2 times the execution time with optimal dynamic allocation. This is useful, for example for evaluating static allocation algorithms, since the problem of finding self-scheduling allocations is certainly not np-hard.

With slight modifications one can establish results optimally comparing cluster allocation to dynamic allocation ([3] and [5]), optimal ratios for parallel programs with a certain parallelism ([6]), the gain of increasing the number of memory units in simultaneous memory allocation problems ([7]), and so on. The end is not in sight.

References:

- [1] R. L. Graham, *Bounds on Multiprocessing Anomalies*, SIAM Journal of Applied Mathematics, Vol. 17, No 2, March 1969, pp. 416-429.
- [2] H. Lennerstad, L. Lundberg, *An Optimal Execution Time Estimate for Static versus Dynamic Allocation in Multiprocessor Systems*, SIAM Jou. of Comp., to appear.
- [3] H. Lennerstad, L. Lundberg, *Optimal Performance Functions Comparing Process Allocation Strategies in Multiprocessor Systems*, Research Report 3/93, Höskolan i Karlskrona/Ronneby, Sweden (1993).
- [4] L. Lundberg, H. Lennerstad, *An Optimal Bound on the Gain of Using Dynamic versus Static Allocation in Multiprocessor Computers*, Technical Report, Höskolan i Karlskrona/Ronneby, (1992).
- [5] L. Lundberg, H. Lennerstad, *An Optimal Performance Bound on the Gain of Using Dynamic versus Static Allocation in Multiprocessors with Clusters*, Technical Report, Höskolan i Karlskrona/Ronneby, (1993).
- [6] L. Lundberg, H. Lennerstad, *An Optimal Upper Bound on the Minimal Completion Time in Distributed Supercomputing*, Proceedings of the 8th ACM Conference on Supercomputing, Manchester, England, July 1994.

- [7] L. Lundberg, H. Lennerstad, *Bounding the Maximal Gain of Changing the Number of Memory Modules in Multiprocess Computing*, Technical Report, Högskolan i Karlskrona/Ronneby, (1994).