Blekinge Institute of Technology
Research Report No 2002:03

A Method for Bounding the Minimal Completion Time in Multiprocessors

by
Magnus Broberg, Lars Lundberg,
Kamilla Klonowska

Department of Software Engineering and Computer Science
Blekinge Institute of Technology

A Method for Bounding the Minimal Completion Time in Multiprocessors

Magnus Broberg, Lars Lundberg, and Kamilla Klonowska

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Soft Center, S-372 25 Ronneby, Sweden
Phone: +46-(0)457 385822
Fax: +46-(0)457 27125
Magnus.Broberg@bth.se, Lars.Lundberg@bth.se, Kamilla.Klonowska@bth.se

Abstract

The cluster systems used today usually prohibit that a running process on one node is reallocated to another node. A parallel program developer thus has to decide how processes should be allocated to the nodes in the cluster. Finding an allocation that results in minimal completion time is NP-hard and (non-optimal) heuristic algorithms have to be used. One major drawback with heuristics is that we do not know if the result is close to optimal or not.

In this paper we present a method for finding a guaranteed minimal completion time for a given program. The method can be used as a bound that helps the user to determine when it is worth-while to continue the heuristic search. Based on some parameters derived from the program, as well as some parameters describing the hardware platform, the method produces the minimal completion time bound. The method includes an aggressive branch-and-bound algorithm that has been shown to reduce the search space to 0.0004%. A practical demonstration of the method is presented using a tool that automatically derives the necessary program parameters and produces the bound without the need for a multiprocessor. This makes the method accessible for practitioners.

Key words: analytical bounds, minimal completion time, parallel programs, multiprocessors, clusters, processor allocation, branch-and-bound, development tool

1. Introduction

Multiprocessors are often used to increase performance. In order to do this, the program processes have to be distributed over several processors. Finding an efficient allocation of processes to processors can be difficult. Clusters of computers use communication networks to send messages between the processes. In almost all cases it is impossible to (efficiently) move a process from one computer to another, and static allocation of processes is thus essential and an unavoidable aspect of such systems.

Even if we consider shared memory multiprocessors, which are often built using a distributed memory approach, we have to consider the allocation issues. Although the network connecting the processors is of high capacity the time for accessing remote memory is 3 to 10 times longer than accessing local memory [21]. In order to avoid that a process is scheduled on different nodes, and thus make the working set for the process into remote memory accesses, one would like to statically bind/allocate the process to a node in the system. The synchronizations between processes will still be performed remotely.

Finding an allocation of processes to processors that results in minimal completion time is a classic allocation problem that is known to be NP-hard [7]. Therefore, heuristic algorithms have to be used, and this results in solutions that may not be optimal. A major problem with the heuristic algorithms is that we do not know if the result is near or far from optimum, i.e. we do not know if it is worth-while to continue the heuristic search for better allocations.

In this paper we present a method for finding a completion time that, given a certain program, can be achieved. The method can be used as an indicator for the completion time that a good heuristic ought to obtain. The method produces the minimal completion time given some parameters derived from the program as well as some parameters describing the hardware platform. The produced performance bound is optimally tight given the information that we have available about the parallel program and the target multiprocessor.

The result presented here is an extension of previous work [17][18]. The main difference between this result and the previous result is that we now can take network communication time and program granularity into consideration. A practical demonstration of the method is presented at the end of the paper.

2. Definitions and main result

A parallel program consists of a set of sequential processes. The execution of a process is controlled by two synchronization primitives: Wait(Event) and Activate(Event), where Event couples a certain Activate to a certain Wait. When a process executes an Activate on an event, we say that the event has occurred. It may, however, take some time for the event to travel from one processor to another. We call that time the synchronization latency t. If a process executes a Wait on an event which has not yet occurred, that process becomes blocked until another process executes an Activate on the same event and the time t has elapsed. However, if both processes are on the same processor, we assume time for the event to travel to be zero, i.e. zero synchronization latency. A process executing a Wait on an event which has occurred more than t time units before does not become blocked. However, if the event occurred less than t time units ago the process executing a Wait on the event has to block for the remaining part of t, unless both processes reside on the same processor (we assume time for the event to travel to be zero within a processor).

Each process can be represented as a list of sequential segments, which are separated by a Wait or an Activate (see Figure 1). We assume that, for each process, the length and order of the sequential segments are independent of the way processes are scheduled. All processes are created at the start of the execution. Some processes may, however, be initially blocked by a Wait, thus imitating the behaviour that one process creates another process. Under these conditions, the minimal completion time for a program *P*, using a system with k processors, a latency of t and a specific allocation denoted A, is $T(P, k, t, A)$. We further find the minimal completion time for a program *P*, using a system with k processors, a latency of t, as $T(P, k, t) = min_A T(P, k, t, A)$.

The left part of Figure 1 shows a parallel program consisting of three processes (P1, P2, and P3). Sequential processing is represented by a procedure Work, i.e. Work(*x*) denotes sequential processing for x time units. Process P1 cannot start its execution before P2 has started. This dependency is represented with a Wait on event 1 in P1. The right part shows a graphical representation of P and two schedules resulting in minimum completion time for a computer with three ($T(P, 3, t)$) and two processors ($T(P, 2, t)$), respectively. We assume that each parallel program has a well defined start and end, i.e., that there is some code that is always executed first and some other code that is always executed last. The thin slices in the beginning and end of P2 represent such a well defined start and end of the program, no actual execution is performed since there is no corresponding Work. The left part of Figure 1 (two processors) shows local scheduling, which means that two or more processes share the same processor. The local schedule, i.e., the order in which processes allocated to the same processor are scheduled affects the completion time of the program. We assume optimal local scheduling when calculating $T(P, k, t)$.

```
process P1
begin
   Wait(Event_1);
   Work(1);
   Activate(Event_3);
   Work(1);
   Activate(Event_4);
end P1;

process P2
begin
   Activate(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Work(2);
   Wait(Event_4);
   Wait(Event_5);
end P2;

process P3
begin
   Wait(Event_2);
   Work(2);
   Activate(Event_5);
end P3;
```

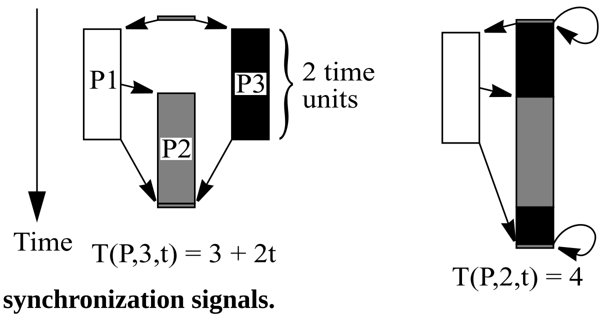


Figure 1: Program *P* with synchronization signals.

For each program P with n processes there is a parallel profile vector V of length n. Entry i in V $(1 \le i \le n)$ contains the fraction of the completion time during which there are i active processes, using a schedule with one process per processor and no synchronization latency. The completion time for a program P with n processes, using a schedule with one process per processor and no synchronization latency is denoted $T(P, n, 0)$. $T(P, n, 0)$ is fairly easy to calculate. In Figure 1, the completion time for the parallel program, using a schedule with one process per processor and no synchronization latency is 3 time units, i.e. $T(P, n, 0) = 3$. During time unit one there are two active processes (P1 and P3), during time unit two there are three active processes (P1, P2, and P3), and during time unit three there is one process (P2), i.e. $V = (1/3, 1/3, 1/3)$. Different parallel programs may, obviously, yield the same parallel profile vector.

For each program P there is a granularity, denoted z, that represents the program's synchronization frequency. By adding the work time of all processes in program P, disregarding synchronization, we obtain the total work time of that program. The number of synchronization signals in program P is divided by the total work time for a program in order to get the granularity, z. In the example in Figure 1 the granularity equals $5/6$.

The completion time is affected by the way processes are allocated to processors. Finding an allocation which results in minimal completion time is NP-hard. However, in this paper we will show that a function $p(n, k, t, z, V)$ can be calculated such that for any program P with n processes, granularity z, and a parallel profile vector V: $min_A(T(P, k, t, A)) = T(P, k, t) \le p(n, k, t, z, V)T(P, n, 0)$. The function $p(n, k, t, z, V)$ is optimal in the sense that for at least some program P, with n processes, granularity z, and a parallel profile vector V: $min_A(T(P, k, t, A)) = T(P, k, t) = p(n, k, t, z, V)T(P, n, 0)$. Consequently, for all programs P with n processes, granularity z, and a parallel profile vector V: $p(n, k, t, z, V) = max_P(T(P, k, t)/T(P, n, 0))$.

The outline for this paper is found in Figure 2. In Section 3 we will show some transformation techniques that allow us to split the program into two parts. The first part includes all the execution time and the other consists of synchronizations only. Then the two parts will be examined resulting in an analytical model for each part in Section 4 and Section 5, respectively. In Section 6 we combine the results from the two parts into a single result that covers the whole program. Following that there is a section where we practically demonstrate the use of this method using a tool. Towards the end we have some discussion and related work. Finally, we have the conclusions.

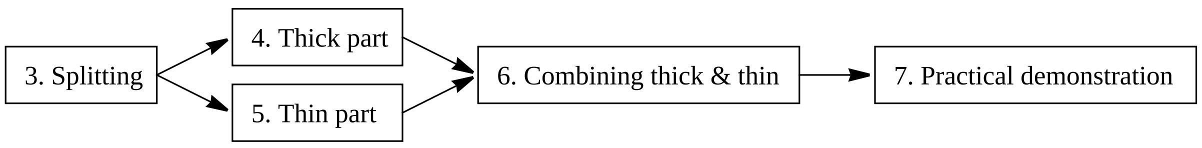


Figure 2: The outline of this paper.

3. Splitting the program into one thick part and one thin part

In this section we will look at techniques, that transform a program P into two parts, one thin part only consisting of synchronizations, and the other part consisting of all the execution time. We will then discuss the thick and thin parts separately in Section 4 and Section 5, respectively.

3.1. Obtaining P' as *m* identical copies of program P

We construct a program P' that is m copies of program P.

Lemma 1: $T(P, k, t)/T(P, n, 0) = T(P', k, t)/T(P', n, 0)$.

Proof: Having m $(m > 1)$ copies of program P, means that we multiply both $T(P, k, t)$ and $T(P, n, 0)$ by m: $T(P, k, t)/T(P, n, 0) = mT(P, k, t)/mT(P, n, 0) = T(P', k, t)/T(P', n, 0)$.
■

Figure 3 shows the transformation of the program P into m copies of this program, denote as P'. Program P (left part in the figure) consists of execution time and synchronization signals.

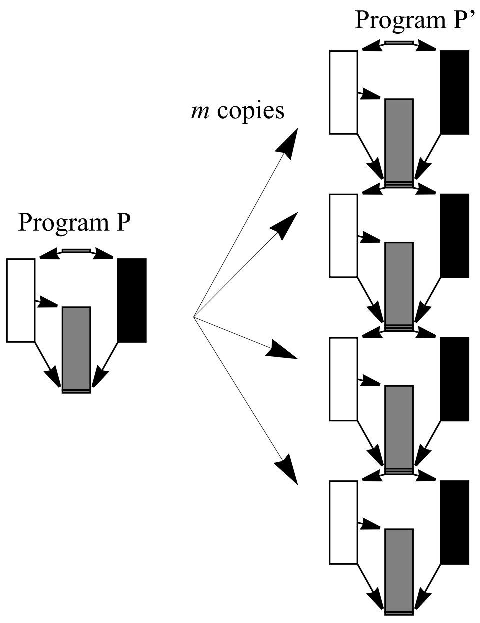


Figure 3: Transformation *P* into *P'*.

3.2. Replacing four copies of a program with three new programs

We prolong each time unit $x, x > 0$ of each process in program P by Δx and get program P'. Program P' is then transformed into P'' in the same way, i.e. after the transformation each time unit $x + \Delta x$ is prolonged with Δx.

In the case with one process per processor and no communication cost the difference of the completion time of $T(P'', n, 0) - T(P', n, 0) = T(P', n, 0) - T(P, n, 0) = \Delta x\, T(P,n,0)/x$.

The situation with synchronization latency is more difficult. Since the synchronization cost in this case is not zero, the differences after prolongation are not always equal, because we do not prolong the synchronizations themselves. Let ΔL denote the difference in length between P and P'. If we denote the length of program P with L, the length of P' will be $L' = L + \Delta L$. In the same way we create P'' with a difference between P' and P'' called $\Delta L'$. The length of P'' will then be the length of $L'' = (L + \Delta L) + \Delta L'$ (see Figure 4).

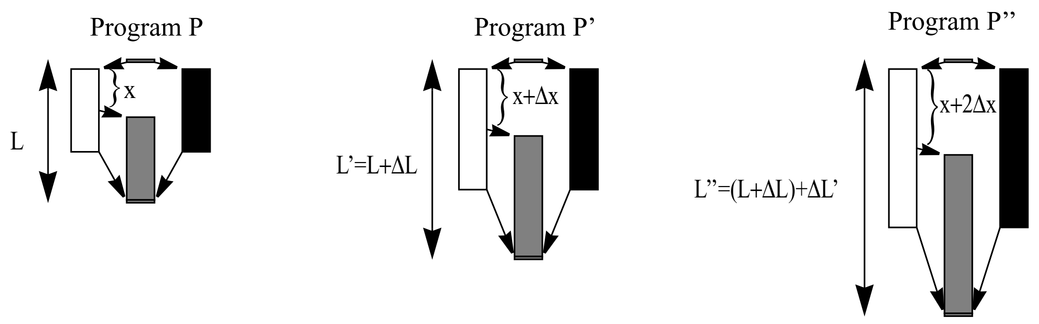


Figure 4: The transformation of the program P by prolongation of the processes.

In order to discuss the effects of local scheduling separately, we will assume that there is only one process per processor. We will relax this restriction at the end of this section (Section 3.2). The *critical path* is defined as the longest path from the start to the end of the program following the synchronizations. In the case when two (or more) paths are the longest, the path with the minimum number of arrows (synchronizations) is the critical path.

Let $arr(P)$ be a number of arrows in the critical path in the program P, and $arr(P')$ be the number of arrows in the critical path in P' (i.e. P after the prolongation).

Lemma 2: $arr(P) \geq arr(P')$.

Proof: Suppose that $arr(P) = x, (x \geq 0)$ and that it in program P there is another path that consists of more than x arrows. Because we prolong the processes, the path with more arrows (and thus less execution) increases slower than the critical path. Consequently, the path with more arrows never can be longer than the critical path.

■

Then of course: $arr(P') \geq arr(P'')$.

When we prolong a program the critical path may change its way (see Figure 5). This happens when path two is longer than path one, and path two has less execution (and thus more synchronizations) than path one. As a consequence of the prolongation, path one will grow faster than path two. At a prolongation of a given Δx the resulting program P' with the paths one and two will have the same length. Adding yet a Δx will yield program P'' where path one is the longest path.

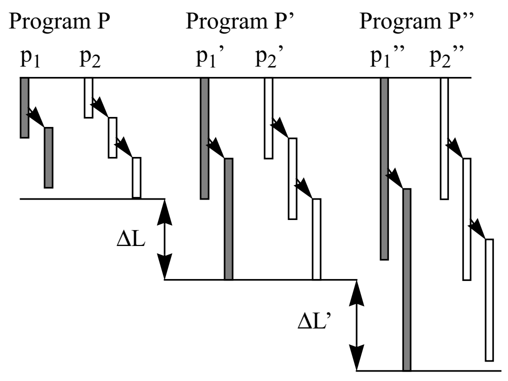


Figure 5: Path p_1 grows faster than p_2 when we add Δx.

Theorem 1: $\Delta L \le \Delta L'$.

Proof: Let $E_1 = y_1 + arr(E_1)t$ be the length of path one, where y_1 is the sum of the lengths of the segments in path one, $arr(E_1)$ is the number of arrows with the communication cost t. Let $E_2 = y_2 + arr(E_2)t$ be the corresponding length for path two. Further, let $y_1 > y_2$ and path two be the critical path, i.e. $E_1 < E_2$. Then let $E_1' = y_1\frac{x + \Delta x}{x} + arr(E_1)t$ and $E_2' = y_2\frac{x + \Delta x}{x} + arr(E_2)t$. Let also $E_1'' = y_1\frac{x + 2\Delta x}{x} + arr(E_1)t$ and $E_2'' = y_2\frac{x + 2\Delta x}{x} + arr(E_2)t$. There are three possible alternatives (provided that there are only these two paths in the system):

- $E_1' < E_2'$ and $E_1'' < E_2''$. In this case $\Delta L = \Delta L'$, the critical path does not change.
- $E_1' < E_2'$ and $E_1'' \ge E_2''$. In this case $\Delta L < \Delta L'$, since path one grows faster than path two when we add Δx.
- $E_1' \ge E_2'$ and $E_1'' > E_2''$. In this case $\Delta L < \Delta L'$, since path one grows faster than path two when we add Δx.

If there are more than two paths in the program, we may have another path (besides path one and two) that becomes the critical path in *P*”. In this case we get $\Delta L < \Delta L'$, since the new path must contain more processing (and less synchronizations) and thus grow faster than paths one and two when we add Δx.

■

According to Theorem 1 we have:

Theorem 2: $2L' \le L + L''$.

Proof: $2L' = L + \Delta L + L + \Delta L \le L + \Delta L + L + \Delta L' = L + (L + \Delta L + \Delta L') = L + L''$.

■

This means that the length of two copies of P' is less than or equal to the length of P plus the length of P''.

We will now look at the case where there can be more than one process on each processor. Let programs *P*, *P’* and *P”* be programs where there are more than one process on some processors; *P*, *P’* and *P”* are identical except that each work-time in *P”* is twice the corresponding work-time in *P’*, and all work-times are zero in *P*. Consider an execution of *P”* using allocation *A* and optimal local scheduling. Let *Q”* be a program where we have merged all processes executing on the same processor into one process. Figure 6 shows how processes *P*2” and *P*3” are merged into process *Q*2”. Let *Q’* be the program which is identical to *Q”* with the exception that each work-time is divided by two. Let *Q* be the program which is identical to *Q”* and *Q’* except that all work-times are zero.

From Theorem 2 we know that $2T(Q', k, t, A) \le T(Q, k, t, A) + T(Q'', k, t, A)$. We use the same allocation *A* for both *P”* and *Q”*. However, since there are less processes in *Q”* we ignore the allocation of non-existing processes in *Q”*, i.e. each process in *Q”* is allocated to a processor of its own. From the definition of *Q”* we know that *T*(*P”*, *k*, *t*, *A*) = *T*(*Q”*, *k*, *t*, *A*). Since the optimal order in which the processes allocated to the same processor (i.e. the optimal local schedule) may not be the same for *P’* and *P”* we know that $T(P', k, t, A) \le T(Q', k, t, A)$, since *Q’* by definition is created with the optimal local scheduling of *P”*.

```
process P1’’
begin
   Wait(Event_1);
   Work(1);
   Activate(Event_3);
   Work(1);
   Activate(Event_4);
end P1’’;
```

```
process P2’’
begin
   Activate(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Work(2);
   Wait(Event_4);
   Wait(Event_5);
end P2’’;
```

```
process P3’’
begin
   Wait(Event_2);
   Work(2);
   Activate(Event_5);
end P3’’;
```

```
process Q1’’
begin
   Wait(Event_1);
   Work(1);
   Activate(Event_3);
   Work(1);
   Activate(Event_4);
end Q1’’;
```

```
processQ2‘’
begin
   Activate(Event_1);
   Activate(Event_2);
   Wait(Event_2);
   Work(1+t);
   Wait(Event_3);
   Work(2);
   Work(1-t);
   Activate(Event_5);
   Wait(Event_4);
   Wait(Event_5);
end Q2’’;
```

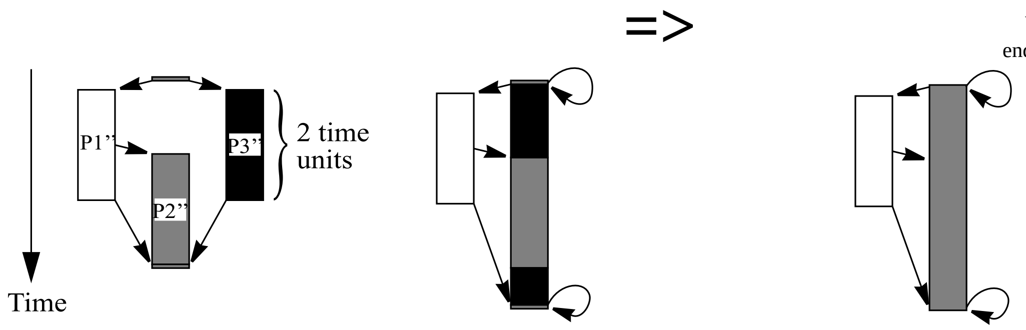


Figure 6: Transforming a program *P”*, allocated to two processors, into a program *Q”* with one process per processor.

Consider now a program *R*, such that the number of processes in *R* is equal to the number of processes in *P*, and such that *R* also has zero work-time (i.e. *R* contains only synchronizations, just like *P*). The number of synchronizations between processes R*i* and R*j* in program *R* is twice the number of synchronizations between P*i* and P*j* in program *P*, i.e. there is always an even number of synchronizations between any pair of processes in *R*. All synchronizations in *R* must be executed in sequence (see Figure 7). We know that it is always possible to form such a sequence, since there is an even number of synchronizations between any pair of processes.

Sequential execution of synchronizations obviously represents the worst case, and local scheduling does not affect the execution time of a sequential program. We thus know that $2T(P, k, t, A) \le T(R, k, t, A)$ and $2T(Q, k, t, A) \le T(R, k, t, A)$.

Consequently,
$$4T(P', k, t, A) \le 4T(Q', k, t, A) \le 2T(Q'', k, t, A) + 2T(Q, k, t, A) \le 2T(P'', k, t, A) + T(R, k, t, A).$$

```
process P1
begin
   Wait(Event_1);
   Activate(Event_3);
   Activate(Event_4);
end P1;
```

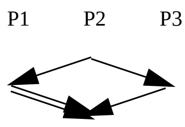
```
process P2
begin
   Activate(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Wait(Event_4);
   Wait(Event_5);
end P2;
```

```
process P3
begin
   Wait(Event_2);
   Activate(Event_5);
end P3;
```

```
process R1
begin
   Wait(Event_1);
   Activate(Event_2);
   Wait(Event_3);
   Activate(Event_4);
   Wait(Event_5);
   Activate(Event_6);
end P1;
```

```
process R2
begin
   Activate(Event_1);
   Wait(Event_2);
   Activate(Event_3);
   Wait(Event_4);
   Activate(Event_5);
   Wait(Event_6);
   Activate(Event_7);
   Wait(Event_8);
   Activate(Event_9);
   Wait(Event_0);
end P2;
```

```
process R3
begin
   Wait(Event_7);
   Activate(Event_8);
   Wait(Event_9);
   Activate(Event_0);
end P3;
```



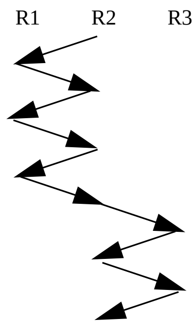
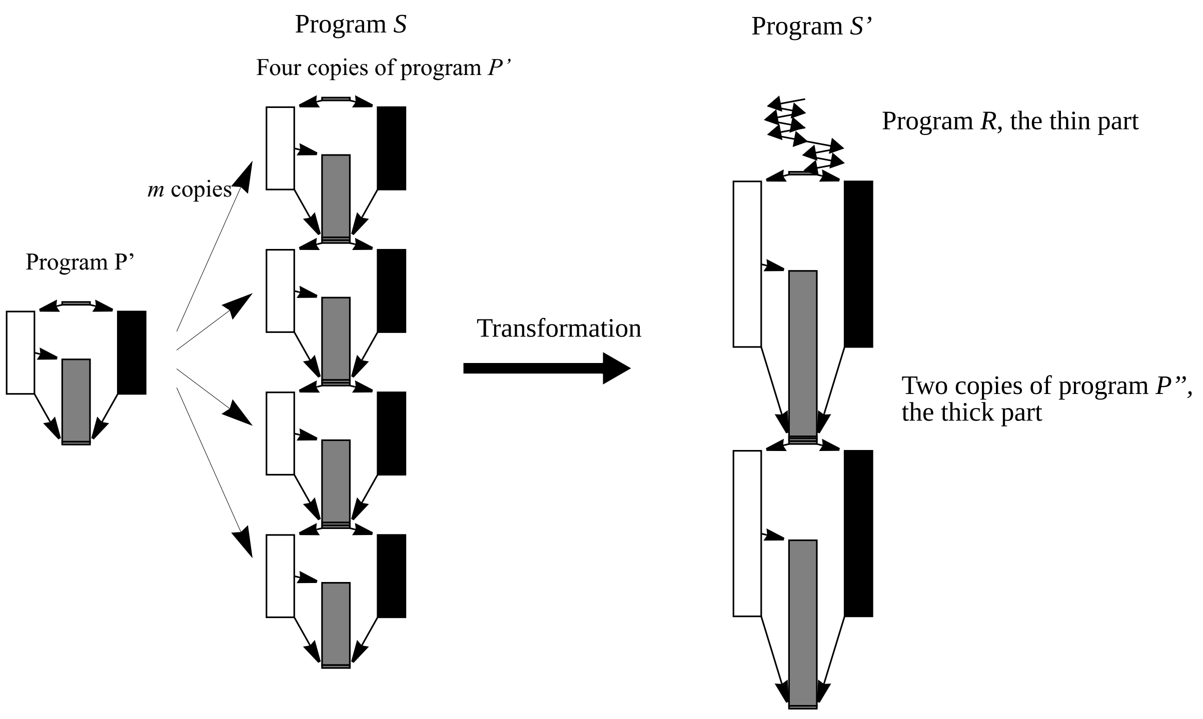


Figure 7: Transforming program *P* into program *R*.

3.3. Transforming program *P* into a program with a thick and a thin section

In this section we describe how to transform an arbitrary program into a program with one part consisting of synchronization only and the other part with all the execution time. We start with an arbitrary program *P'*. First we create *m* copies of *P'*, where $m = 2^x$, for some integer *x* ($x \geq 2$). We then combine the *m* copies in groups of four and transform the four copies of *P'* to two programs *P"* and one program *R*. From the discussion above we know that $4T(P', k, t, A) \leq 2T(P'', k, t, A) + T(R, k, t, A)$ for any allocation *A,* i.e. $4T(P', k, t) \leq 2T(P'', k, t) + T(R, k, t)$. Note that $4T(P', n, 0) = 2T(P'', n, 0) + T(R, n, 0)$, and that the parameters *n, V* and *z* are invariant in this transformation. We now end up with 2^{x-1} programs *P"*. Again we combine these 2^{x-1} *P"* programs in groups of four and use the same technique and end up with 2^{x-2} programs *P'"* (with twice the execution time compared to *P"*) and one program *R*. We repeat this technique until there are $2^{x-1} - 1$ thin *R* programs and two very thick programs (i.e. all the execution time is concentrated to two copies of the original program). By selecting a large enough m, we can neglect the synchronization times in these two copies. This is illustrated in Figure 8, where we demonstrate it for $m = 4$. Note that the execution time using k processors may increase due to this transformation, whereas $T(P, n, 0)$ is unaffected.

Thus, we are able to transform a program into two parts: a thin part consisting only of synchronizations, and the other part consisting of all the execution time. We will then discuss the thick and thin parts separately in Section 4 and Section 5, respectively.



Program *S*
Four copies of program *P'*
Program *S'*
Program *R*, the thin part
m copies
Program P'
Transformation
Two copies of program *P"*, the thick part

Figure 8: The transformation of *m* copies of the program P into a thick and a thin part. The transformation guarantees that $T(P', k, t)/T(P', n, 0) = T(S, k, t)/T(S, n, 0) \leq T(S', k, t)/T(S', n, 0)$.

4. The thick part

In this section we will deal with the thick part of the program. The thick part has the nice property that we can assume that the synchronization time (t) is arbitrarily small, i.e. we can assume $t = 0$. We can do this since selecting a large enough m in Section 3 will cause the synchronization time in the thick part to be negligible because virtually all synchronizations will be in the thin part. We are thus also free to introduce a constant number of new synchronizations without any problems.

These properties of the thick part will allow us to first transform the thick part into a matrix consisting of zeros and ones. Using this matrix representation we can then find the worst possible program of all programs with n processes and a parallel profile V. Finally, we will show a formula that we can use when calculating the execution time of this worst case program given an allocation. More elaborate proof and discussion concerning the transformations in this section can be found in [11] and [12].

4.1. Transforming *P* into *Q*

We consider $T(P, n, 0)$. This execution is partitioned into m equally sized time slots in such a way that process synchronizations always occur at the end of a time slot. In order to obtain Q we add new synchronizations at the end of each time slot. These synchronizations guarantee that no processing done in slot r, $(1 < r \leq m)$, can be done unless all processing in slot $r - 1$, has been completed. The net effect of this is that a barrier synchronization is introduced at the end of each time slot. Figure 9 shows how a program P is transformed into a new program Q by this technique.

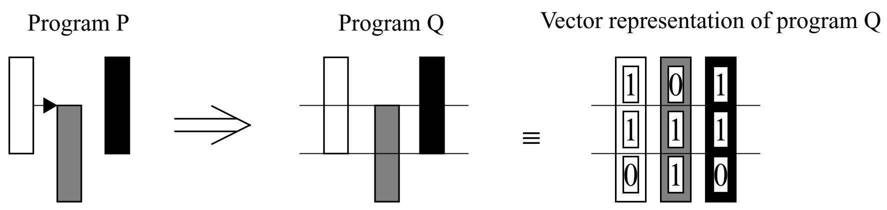


Figure 9: The transformation of program P

The synchronizations in Q form a superset of the synchronizations in P, i.e. $T(P, k, 0) \le T(Q, k, 0)$ (in the thick part of the program we assume that $t = 0$). Consequently, $T(P, k, 0)/T(P, n, 0) \le T(Q, k, 0)/T(Q, n, 0)$. It is important to note that we obtain the same parallel profile vector for both P and Q.

In order to simplify the discussion we introduce an equivalent representation of Q, called the vector representation (see Figure 9). In this representation, each process is represented as a binary vector of length m, where m is the number of time slots in Q, i.e. a parallel program is represented as n binary vectors. In some situations we treat these vectors as one binary $m \times n$ matrix, where each column corresponds to a vector and each row to a time slot.

From now on we assume unit time slot length, i.e. $T(Q, n, 0) = m$. However, $T(Q, k, 0) \ge m$, because if the number of active processes exceeds the number of processors on some processor during some time slot, the execution of that time slot will take more than one time unit.

With P_{nv} we denote the set of all programs with n processes and a parallel profile V. From the definition of the parallel profile vector, V, we know that if $Q \in P_{nv}$, then $T(P, n, 0)$ must be a multiple of a certain minimal value m_v, i.e. $V = \left(\frac{x_1}{m_v}, \frac{x_2}{m_v}, \dots, \frac{x_n}{m_v}\right)$ where $T(P, 1, 0) = xm_v$, for some positive integer x.

Programs in P_{nv} for which $T(Q, n, 0) = m_v$ are referred to as minimal programs. For instance, the program in Figure 1 is a minimal program for the parallel profile vector $V = (1/3, 1/3, 1/3)$.

We are now able to handle programs as a binary matrix. Each column in the matrix represent one process, and the rows are independent of each others.

4.2. Transforming *Q* into *Q'*

We start by creating $n!$ copies of Q. The vectors in each copy are reordered in such a way that each copy corresponds to one of the $n!$ possible permutations of the n vectors. Vector number v $(1 \le v \le n)$ in copy number c $(1 \le c \le n!)$ is concatenated with vector number v in copy $c + 1$, thus forming a new program Q' with n vectors of length $n!m$. The execution time from slot $1 + (c - 1)m$ to cm $(1 \le c \le n!)$ cannot be less than $T(Q, k, 0)$, using k processors. Consequently, $T(Q', k, 0)$ cannot be less than $n!T(Q, k, 0)$. Since, $T(Q', n, 0) = n!m$, we know that $T(Q, k, 0)/T(Q, n, 0) = T(Q, k, 0)/m = n!T(Q, k, 0)/(n!m) \le T(Q', k, 0)/T(Q', n, 0)$. It is important to note that we obtain the same parallel profile for both Q' and Q.

The n vectors in Q' can be considered as columns in a $n!m \times n$ matrix. Reordering the rows in this matrix affects neither $T(Q', k, 0)$ nor $T(Q', n, 0)$. The rows in Q' can be reordered into n groups, where all rows in the same group contain the same number of ones (some groups may be empty). Figure 10 shows how program Q is transformed into a new program Q'.

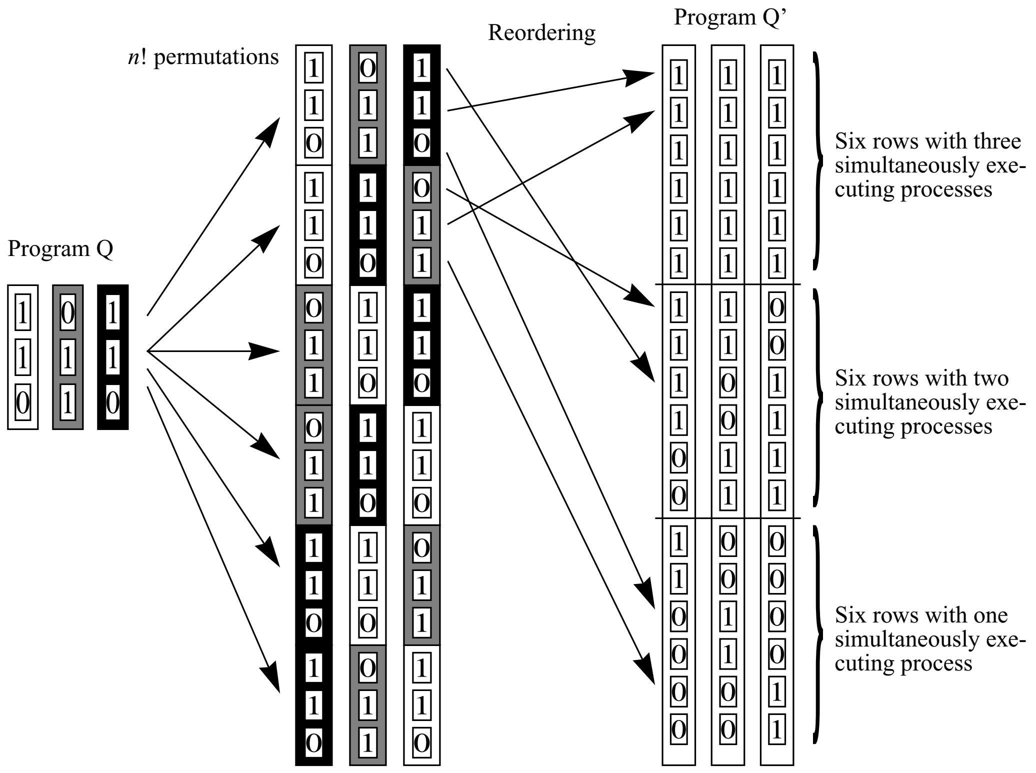

Program Q
n! permutations
Reordering
Program Q'
Six rows with three simultaneously executing processes
Six rows with two simultaneously executing processes
Six rows with one simultaneously executing process

Figure 10: The transformation of vector representation of program Q into Q'.

Due to the definition of minimal programs, we know that all minimal program Q result in the same program Q_m'. In Figure 10 we have a situation where $Q_m' = Q'$. If $Q2$ $(Q2 \in P_{nv})$ is a program for which $T(Q2, n, 0) = xm_v, (x > 1)$, then the corresponding program $Q2'$ is identical with Q_m', except that each row in Q_m' is duplicated x times. Creating x identical copies of each row does not affect the ration $T(Q', k, 0)/T(Q', n, 0)$. Consequently, all programs Q' in P_{nv} can be mapped onto the same Q_m'. We have now found the worst possible program Q_m' ($Q_m' \in P_{nv}$).

4.3. Allocation properties of the thick part

The completion time of Q_m' is not affected by the identity of the processes allocated to different processors, because if vector p_1 is allocated to processor A and vector p_2 is allocated to processor B, then moving p_1 to B and p_2 to A is equivalent to reordering the rows in Q_m'. Obviously, reordering the rows does not affect the completion time. Consequently, the completion time of Q_m' is affected only by the number of vectors allocated to the different processors.

Theorem 3: If there are n_1 vectors allocated to processor A and n_2 vectors to processor B and $n_1 < n_2$, the completion time cannot increase if we move one vector from processor B to processor A.

Proof: We order the vectors in Q_m' in such a way the vectors 1 to n_1 are allocated to processor A and vectors $n_1 + 1$ to $n_1 + n_2$ are allocated to processor B, then we prove that moving vector $n_1 + 1$ to processor A does not increase the completion time.

If vector $n_1 + 1$ has a zero in slot r $(1 \le r \le n!m_v)$, the contribution to the completion time from row r will not be affected by moving vector $n_1 + 1$ from processor B to processor A. Consequently, we have only to consider rows for which the corresponding slot is one in vector $n_1 + 1$. Moreover, if the number of ones in positions 1 to n_1 is smaller than the number of ones in positions $n_1 + 1$ to $n_1 + n_2$, the contribution to the completion time from row r does not increase.

Q_m' contains all permutations. Consequently, for each row r such that position $n_1 + 1$ contains a one, and there are at least as many ones in positions 1 to n_1 as in positions $n_1 + 1$ to $n_1 + n_2$, there exists an (n_1, n_2)-permutation r'. An (n_1, n_2)-permutation of a row is obtained by switching items i and $n_1 + n_2 + 1 - i$ $(1 \le i \le n_1)$, see Figure 11.

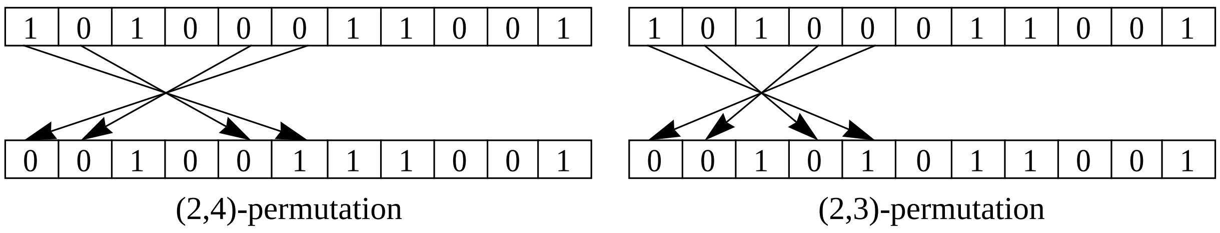


Figure 11: One (2,4)-permutation and one (2,3)-permutation of the same row.

If the contribution to the completion time from row r increases from h to $h + 1$ when we move vector $n_1 + 1$ from processor B to processor A, there must be h ones in position 1 to n_1 in row r. In that case we know that the symmetry of the (n_1, n_2)-permutation guarantees that there are at least $h + 1$ ones in positions $n_1 + 1$ to $n_1 + n_2$ in row r'. Therefore, the contribution to the completion time from r' will decrease with one when moving vector $n_1 + 1$ from processor B to processor A. Consequently, the completion time of Q_m' cannot increase if we move one vector from processor B to processor A.

■

As a consequence, an allocation of Q_m' results in shorter completion time the more evenly the processes are spread out on the processors. Also the identity of the processes is of no importance, and we can thus order the n vectors in some arbitrary order. In fact, the minimal completion time is obtained by allocating vector number $c + ik$ to processor c $(1 \le c \le k, 0 \le i \le \lfloor n/k \rfloor)$.

4.4. Calculating the thick section

The most obvious way of calculating the thick part is to generate the $n!m_v \times n$ matrix containing all permutations. This is however extremely inefficient. We will in this section show how to calculate the thick part in an analytical and generic way for any allocation of processes to processors.

The actual allocation is specified using an ordered set. The notation used for an ordered set A is that the number of elements in the ordered set is a, thus $|A| = a$. Also, the *i:th* element of the ordered set is denoted a_i, thus $A = (a_1, \ldots, a_a)$.

First we will show how we handle the case when the number of processes executing simultaneously changes during run-time. Then we will look at how to calculate the execution time of the case with a fixed number of processes executing simultaneously. In order to perform those calculation we use another function. At the end of this section we will demonstrate the most fundamental parts of the functions below using a small example.

4.4.1 *s*(*A*, *V*)

This function is used to handle the case when the number of processes executing simultaneously changes during run-time, indicated by the parallel profile V. This is illustrated by *Q*' in Figure 10. In this figure we have a parallel profile $V = (6/18, 6/18, 6/18) = (1/3, 1/3, 1/3)$. For each entry in this vector we count the ones (using function f described in Section 4.4.2) and use the parallel profile (V) as weight when adding them together. As found in the previous section the allocation A is independent of the process identity, thus the allocation A only shows the *number* of processes allocated to each processor, where A is decreasing, i.e. $a_i \geq a_{i+1}$ for $1 \leq i < a$. Note that we have the parameters n and k implicit in the vectors, $n = v = \sum_{i=1}^{a} a_i$ and $k = a$.

The binary matrix in Figure 10 can be divided into *n* parts such that the number of ones in each row is the same for each part. The relative proportion between the different parts is determined by the parallel profile vector. The function *s(A,V)* is then obtained as the weighted sum of these parts, i.e., $s(A, V) = \sum_{q=1}^{n} \frac{v_q f(A, q, 0)}{\binom{n}{q}}$.

4.4.2 *f*(*A*, *q*, *l*)

This function calculates how long it takes to execute a program with n processes and $\binom{n}{q}$ rows where q processes, and only q, are executing simultaneously using an allocation A, compared to executing the program using one processor for each process. This is done by counting the maximum number of ones for each processor for all rows. The function f is recursively defined and divides the allocation A into smaller sets, where each set is handled during one (recursive) call of the function. The function uses another function later found in Section 4.4.3. The function f will be discussed in detail in Section 4.4.4 using an example.

$f(A, q, l)$ uses q for the number of ones in a row. The l denotes the maximum number of ones in any previous set in the recursion. Note that we have the parameters n and k implicit in the vector, $n = \sum_{i=1}^{a} a_i$ and $k = a$. From the start $l = 0$. We also have $w = a_1$, $d = |\{a_x, \ldots\}|$ ($1 \leq x \leq a$ and $a_x = a_1$) and $b = n - wd$.

If $b = 0$ then $f(A, q, l) = \sum_{l_1 = max(0, \lceil q/d \rceil)}^{min(q, w)} \pi(d, w, q, l_1) max(l_1, l)$, otherwise:

$$f(A, q, l) = \sum_{l_1 = max\left(0, \left\lceil \frac{q-b}{d} \right\rceil\right)}^{min(q, w)} \sum_{i = max(l_1, q-b)}^{min(l_1 d, q)} \pi(d, w, i, l_1) f(A - \{a_1, \ldots, a_d\}, q - i, max(l_1, l)).$$

4.4.3 $\pi(k, w, q, l)$

$\pi(k, w, q, l)$ [11] is a help function to f and denotes the number of permutations of q ones in kw slots, which are divided into k sets with w slots in each, such that the set with maximum number of ones has exactly l ones. $\pi(k, w, q, l) = 0$ if $q < l$ or if $q > kl$, otherwise if $k = 1$ then $\pi(k, w, q, l) = \binom{w}{l}$, otherwise it is given by:

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

Here, the sum is taken over all sequences of non negative integers $I = \{i_1, \ldots, i_{k-1}\}$, which are decreasing, i.e. $i_j > i_{j+1}$ for all $j = 1, \ldots, k-2$, and for which $i_1 \le min(w, l)$ and $\sum_{j=1}^{k-1} i_j = q - l$. The functions $a(I, j)$ and $b(I)$ are defined in the following way:

$a(I, j)$ = the number of occurrences of the *j:th* distinct integer in I.
$b(I)$ = the number of distinct integers in I.

More detailed proofs and discussions for how the functions in this subsection (Section 4.4.3) are obtained can be found in [11] and [12].

4.4.4 A guide through the most fundamental formula in the thick part

In this section we will focus on the function f, since function s is quite simple and the function π is already described in [11] and [12]. The example we will use have the following parameters $A = (2, 2, 1)$ and $q = 3$. This means that we have five processes of which three, and only three, are running simultaneously. We also have three processors, two which are assigned two processes each, and one processor with one process assigned to it. All possible permutations of this system is found in Table 4, where a one indicates that the process is running and a zero that the process is blocked, thus we have the vector representation found in Figure 9. What is more included in the table is the execution time required to execute each row (the left-most column). This column is calculated by finding the processor with the largest number of ones. For example, in row 1 processor A has to execute both process 1 and 2, which means that it will spend 2 time units of execution. Processor B has only process 3 to execute (process 4 has a zero), thus it will spend 1 time unit. Thus the time for executing this row is two time units and is determined by processor A. Another case is row 5, where all three processors only have a single one each, thus the time for executing that row is one time unit. The total time for executing the program is given by adding the execution time per each row, resulting in 16 time units.

Table 4: The permutations of the system $A = (2, 2, 1)$ and $q = 3$.

Row number	Processor A		Processor B		Processor C	Execution time per row
	Process 1	Process 2	Process 3	Process 4	Process 5	
1	1	1	1	0	0	2
2	1	1	0	1	0	2
3	1	1	0	0	1	2
4	1	0	1	1	0	2
5	1	0	1	0	1	1
6	1	0	0	1	1	1
7	0	1	1	1	0	2
8	0	1	1	0	1	1
9	0	1	0	1	1	1
10	0	0	1	1	1	2

We will now briefly describe the basic structure of function f. The allocation ($A = (2, 2, 1)$) is split into sets of processors with equal number of processes assigned to them. Remember that A is decreasing. In our example we have the sets of $(2, 2)$ and (1). Each such set fits into the function π and will be handled separately through the recursion. The variable d is the length of such a set and w is the number of processes in each element in the set, thus dw is the number of processes in the set. The variable b is the number of processes in the remaining part of the allocation, thus representing the maximum number of ones that possibly can be fit into the remaining part of the allocation. The variable l_1 iterates over all possible number of ones that can fit in each processor. Also the variable i iterates over all possible number of ones that can be contained in the whole set. The variable l holds the maximum number of l_1 through the recursion and all the sets. The variable l is set to zero for the first call to function f for the only reason that it will not be larger than any of the l_1 in the sets.

In the following sections we will go through the simple example starting in Section 4.4.5 with the first call to function f with the initial parameters.

4.4.5 *f*(*A*=(2, 2, 1), *q*=3, *l*=0)

From the incoming parameters we can conclude that $n = 5$, $w = 2$, $d = 2$, and $b = 1$. Since $b \neq 0$ the second part of the function is used. Further the variable $l_1 = 1 \ldots 2$. For each iteration over l_1 we have:

- $l_1 = 1$: The variable $i = 2 \ldots 2 = 2$, thus we have the call to function $\pi(2, 2, 2, 1) = 4$ and the (recursive) call to $f((1), 1, 1) = 1$ (see Section 4.4.6 for how this is calculated). The resulting value is then $4 \cdot 1 = 4$.

- $l_1 = 2$: The variable $i = 2\ldots3$, for each iteration over i we have:
  - $i = 2$: The call to function $\pi(2, 2, 2, 2) = 2$ and the (recursive) call to $f((1), 1, 2) = 2$ (see Section 4.4.7 for how this is calculated). The resulting value of this iteration is then $2 \cdot 2 = 4$.
  - $i = 3$: The call to function $\pi(2, 2, 3, 2) = 4$ and the (recursive) call to $f((1), 1, 2) = 2$ (see Section 4.4.7 for how this is calculated). The resulting value of this iteration is then $4 \cdot 2 = 8$.

In total $f((2, 2, 1), 3, 0) = 4 + 4 + 8 = 16$. This is what we concluded by hand in Table 4.

4.4.6 *f*(*A*=(1), *q*=1, *l*=1)

From the incoming parameters we can conclude that $n = 1$, $w = 1$, $d = 1$, and $b = 0$. Since $b = 0$ the first part of the function is used. Further the variable $l_1 = 1\ldots1$. For $l_1 = 1$ we have the call to function $\pi(1, 1, 1, 1) = 1$ and $max(l_1, l) = max(1, 1) = 1$. The resulting value of this iteration is then $1 \cdot 1 = 1$. In total $f((1), 1, 1) = 1$.

4.4.7 *f*(*A*=(1), *q*=1, *l*=2)

From the incoming parameters we can conclude that $n = 1$, $w = 1$, $d = 1$, and $b = 0$. Since $b = 0$ the first part of the function is used. Further the variable $l_1 = 1\ldots1$. For $l_1 = 1$ we have the call to function $\pi(1, 1, 1, 1) = 1$ and $max(l_1, l) = max(1, 2) = 2$. The resulting value of this iteration is then $1 \cdot 2 = 2$. In total $f((1), 1, 2) = 2$.

5. The thin part

From Section 3 we know that the thin part of a program for which the ratio $T(P, k, t)/T(P, n, 0)$ is maximized consists of a sequence of synchronizations, i.e. program *R* in Section 3. From Section 4 we know that the identity of the processes allocated to a certain processor does not affect the execution time of the thick part. In the thin part we would like to allocate processes that communicate frequently to the same processor.

Let y_i be a vector of length *i*-1. Entry *j* in this vector indicates the number of synchronizations between processes *i* and *j*. Consider an allocation *A* such that the number of processes allocated to processor *x* is a_x. The optimal way of binding processes to processors under these condition is a binding that maximizes the number of synchronizations within the same processor.

Consider also a copy of the thin part of the program where we have swapped the communication frequency such that process *j* now has the same communication frequency as process *i* had previously and process *i* has the same communication frequency as process *j* had previously. In Figure 12 the original communication vector for P3 is (6,2), meaning that there are six synchronizations between P3 and P1 and two synchronizations between P2 and P3. In the copy we have swapped the communication frequencies of P2 and P3 and we thus have six synchronizations between P2 and P1 and two synchronizations between P2 and P3.

It is clear that the minimum execution time of the copy is the same as the minimum execution time of the original version. The mapping of processes to processors that achieves this execution time may however, not be the same. For instance, if we have two processors and an allocation *A* = (2,1), we obtain minimum completion time for the original version when P1 and P3 are allocated to the same processor, whereas the minimum for the copy is obtained when P1 and P2 share the same processor.

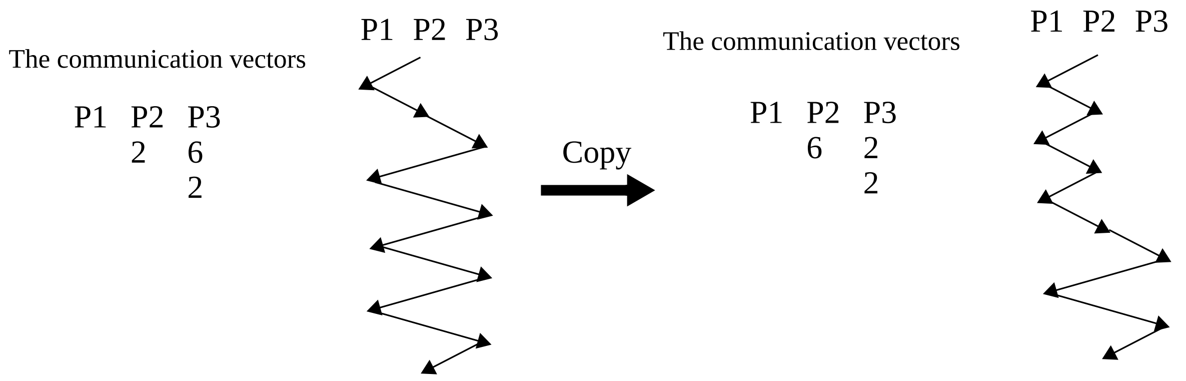


Figure 12: Making a copy where we have swapped the communication frequencies of processes P2 and P3.

If we concatenate the original (which we can call P) and the copy (which we can call Q) we get a new program P' such that $T(P', k, t, A) \geq T(P, k, t, A) + T(Q, k, t, A)$ for any allocation A (all thin programs take zero execution time using a system with one process per processor and no communication delay). By generalizing this argument we obtain the kind of permutation as we had for the thick part (see Figure 10).

These transformations show that the worst case for the thin part occurs when all synchronization signals are sent from all n processes $n - 1$ times - each time to a different process. All possible synchronization signals for n processes equals $n(n-1)$ and all possible synchronization signals between the processes allocated to processor i equals $a_i(a_i - 1)$. Because some processes are executed on the same processor, the communication cost for them equals zero. That means that the number of synchronization signals in the worst-case of program (regarding communication cost) is equal to $n(n-1) - \sum_{i=1}^{k} a_i(a_i - 1)$.

Note that we have the parameters n and k implicit in the vectors, $n = v = \sum_{i=1}^{a} a_i$ and $k = a$.

If $n = 1$ then $r(A, t, V) = 0$, otherwise $r(A, t, V) = \dfrac{n(n-1) - \sum\limits_{i=1}^{k} a_i(a_i - 1)}{n(n-1)} t \sum\limits_{q=1}^{n} (qv_q)$.

6. Combining the thick and thin sections

Combining the thick and the thin section is done by adding the function $s(A, V)$ and $r(A, t, V)$, weighted by the granularity, z. Thus, we end up with a function $p(n, k, t, z, V) = min_A(s(A, V) + zr(A, t, V))$. It should be noted that the allocation A implicitly includes the parameters $n = \sum_{i=1}^{a} a_i$ and $k = a = |A|$. As we can see in the formula we use the granularity, z, as a weight between the thick part and the thin part. In a program with high granularity, i.e. high synchronization frequency, the thin part has larger impact than in a program with low granularity.

As previously shown the thick section is optimal when evenly distributed over the processors, whereas the thin section is optimal when all processes reside on the same processor. The algorithm for finding the minimum allocation A is based on the knowledge about the optimal allocation for the thick and thin part respectively.

6.1. Finding the optimal allocation using allocation classes

The basic idea is to create *classes* of allocations and evaluate them. An allocation class consists of three parts:

- A common allocation for both the thick and thin parts, the allocation here shows the number of assigned processes for the n first processors. The allocation must be decreasing, i.e. the number of assigned processes to processor i must be greater than or equal to the number of assigned processes to processor $i + 1$.
- The allocations for the remaining processors for the thick part, the allocations are evenly distributed. The highest number of processes assigned to a processor must be equal or less than the least number of assigned processes for any processor in the common assignment.
- The allocations for the remaining processors for the thin part, the allocations make use of as few of the remaining processors as possible and with as many processes as possible on each processor. The highest number of processes assigned to a processor must be equal or less than the least number of assigned processes for any processor in the common assignment.

An example of an allocation class with the common part consisting of one processor, $n = 9$ and $k = 4$, is shown in Figure 13(a) where the first part is the common allocation, the upper part is the thick allocation for the remaining 3 processors and the lower part is the thin allocation for the remaining 3 processors. In Figure 13(b) we have the first allocation, with no common allocation at all. In fact this first allocation consider the thick part only and, thus is quite inaccurate. In Figure 13(c) are all the possible allocation classes for this example. By calculating the thick part using the common and thick allocation and the thin part using the common and thin allocation we will get a result that is better than (or equal to) any allocation within that class. This is because we will get an over-optimal result, due to the fact that we have different allocations for the thick and the thin part, which in practice is impossible. Then we take the minimum value of all the classes in Figure 13(c) for the over-optimal allocation.

6.2. Branch-and-bound algorithm

The algorithm for finding an optimal allocation is a classical branch-and-bound algorithm [1]. The allocations can then be further divided, by choosing the class that gave the minimum value and add one processor in the common allocation, lets say in this example (in Figure 13) the class with 5 processors in the common allocation was the minimum. The subclasses are shown in Figure 13(d). All the subclasses will have a higher (or equal) value than the previous class. The classes are organized as a tree structure and the minimum is now calculated over the *leaves* in the tree and a value closer to the optimal is given. By repeatedly selecting the leaf with minimum value and create its subclasses we will reach a situation where the minimum leaf no longer has any subclasses, then we know that we have found the optimal allocation. If we assume that the classes in Figure 13(e) and (f) gives the minimum values we have reached an optimal allocation. When calculating a class we use the common allocation concatenated with the thick allocation as input for the function $s(A_{thick}, V)$ and $r(A_{thin}, t, V)$, where A_{thick} is the common allocation concatenated with the thick allocation and A_{thin} is the common allocation concatenated with the thin allocation. By adding the results (weighted by z) from the two functions we get the value of that leaf ($s(A_{thick}, V) + zr(A_{thin}, t, V)$).

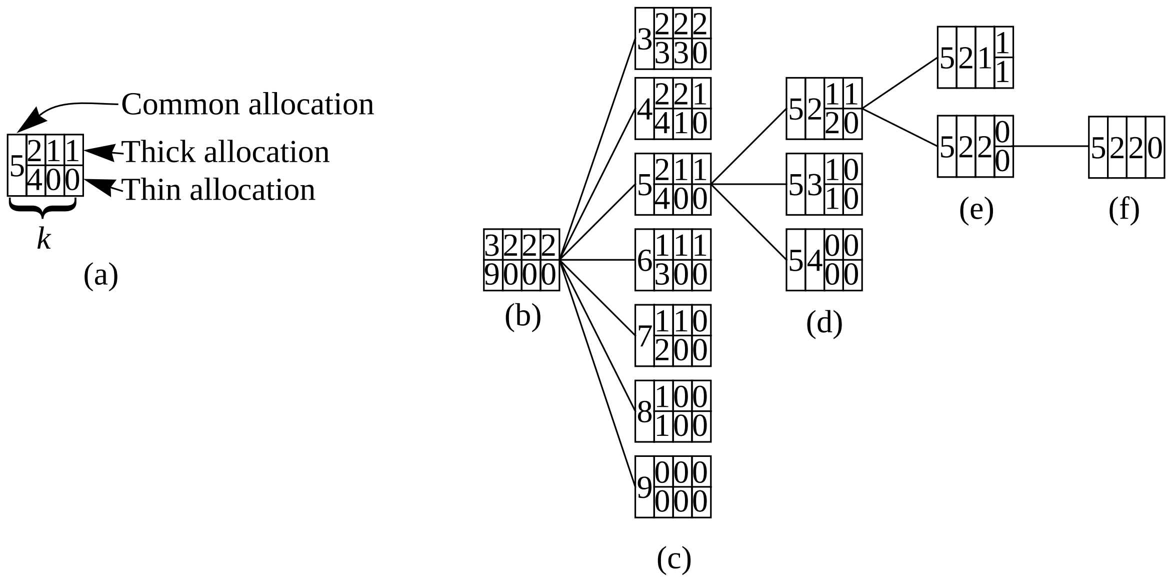


Figure 13: Allocation classes.

6.3. Lazy evaluation

We can further reduce the number of calculations by adopting a lazy evaluation approach. This approach is based on the observation that the value for the thick part (function s) in Figure 13(c) increases as we calculate it for more and more processes on the first processor. This is also shown in Figure 14 for an application with $n = 79, k = 16$. We also know that the value of the thin part is decreasing as we increase the number of processes on the first processor, also shown in Figure 14. The value of an allocation class is, as stated above, the sum of the thick part and the thin part weighted by z, which is shown in Figure 14 for some z.

The lazy evaluation is based on the fact that the thick part is increasing and the thin part is decreasing. We can conclude from Figure 14 that the minimum of the thick and thin curve is at eight processes on the first processor, thus when we reach 31 processes on the first processor we note that the thick part alone is larger than the minimum at 8 processes. We then know that this allocation (and further) can never be the minimum (since the thick part is increasing) regardless how much the thin part decreases. In this case the thin part may become zero since the allocation (9) yields a thin part of zero. We mark all allocations with more than 31 processes on the first processor to have the value of *at least* the value of the *thick part* of 31 processors on the first processor *and* the *thin part* of allocation (9) (which is zero). If we now apply the same technique in the subclasses found in Figure 13(d), we know that the least value of the thin part is found in the allocation $(5, 4)$. This means that the thin part is never lower than in this allocation class, thus we can stop evaluating when the value of the thick part of the current allocation and the thin part of allocation $(5, 4)$ (the allocation with the least thin part) reaches above the minimum. We then mark the remaining allocations with the value of at least the thin part of the current allocation and the thin part of allocation $(5, 4)$.

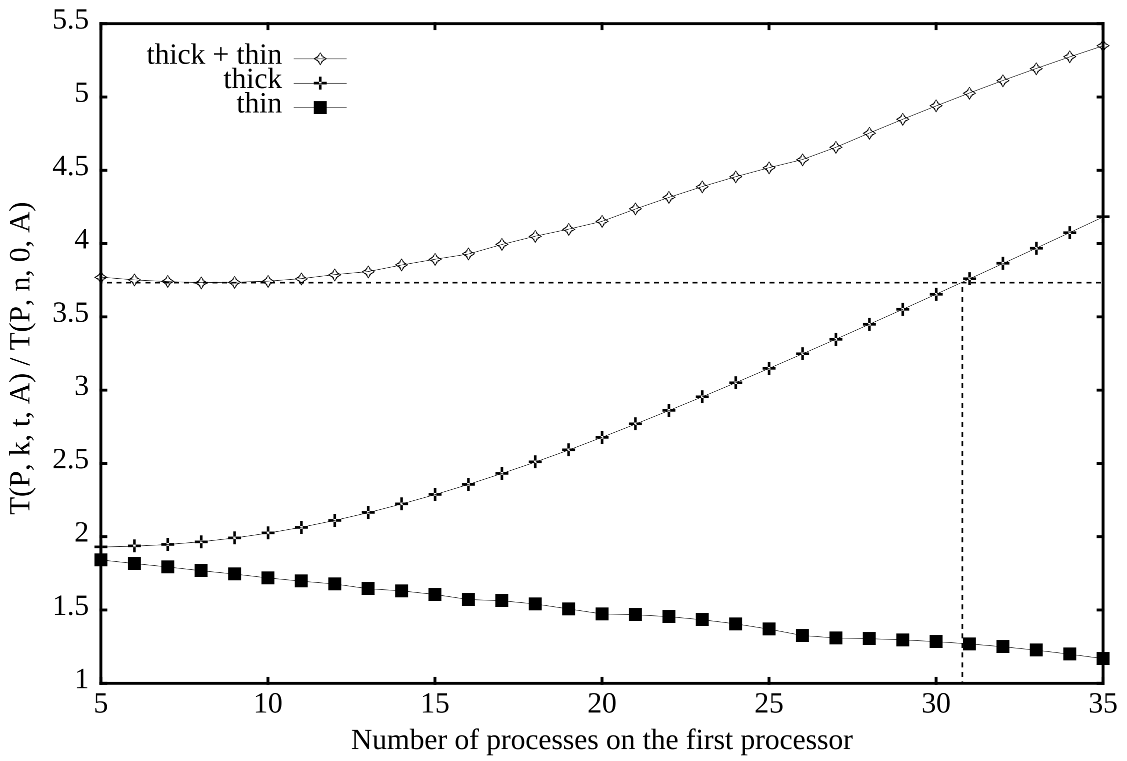


Figure 14: An example showing that the thick part increases and the thin part decreases as the number of processes on the first processor increases.

When we continue (in Figure 14) to make the allocation subclasses to the class with eight processes on the first processor the minimum may increase. At some point the minimum may increase above the value of the thick part at 31 processes on the first processor, then we have to continue to calculate the case with 32 processes and so on until we once again find a thick part that is larger than the minimum. The benefit with this lazy evaluation is that we will not always have to evaluate all allocation classes as later shown in Section 7.2.

The drawback with this algorithm is that it (theoretically) can search through all possible nodes in the tree before finding the optimal solution. However, our practical experience shows that only very few nodes needs to be investigated until the optimal leaf is found, see Section 7.2.

7. Method demonstration

In this section we will demonstrate how the method in this paper can be used in a practical case. Previously we have developed a tool for monitoring multithreaded programs on a uni-processor workstation and then, by simulation, predict the execution of the program on a multiprocessor with any number of processors. The tool is called VPPB (Visualization of Parallel Program Behaviour) [2, 3]. We have modified this tool to extract the program parameters required by our method (n, V, and z). The tool can be given information about the hardware in order to simulate synchronization latency (t) between the k (simulated) processors. We use that ability to compare the results from our method with the simulations.

7.1. Overview of the Tool

The VPPB tool enables the developer to monitor the execution of a parallel program on a uni-processor workstation, and then predicts and visualizes the program behaviour on a simulated multiprocessor with an arbitrary number of processors.

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB tool is shown in Figure 15. The developer writes the multithreaded program (a) in Figure 15, compiles it and an executable binary file is obtained. After that, the program is executed on a single processor workstation. When starting the monitored execution (b), the *Recorder* is automatically inserted *between* the program and the standard thread library. Each time the program uses the routines in the thread library, the call passes through the *Recorder* (c) which records information about the call, i.e., the identity of the calling thread, the name of the called routine, the time the call was made, and other parameters. The *Recorder* then calls the original routine in the thread library. When the execution of the program finishes all the collected information is stored in a file, the *recorded information* (d). The recording is done without recompilation or relinking of the application, making our approach very flexible.

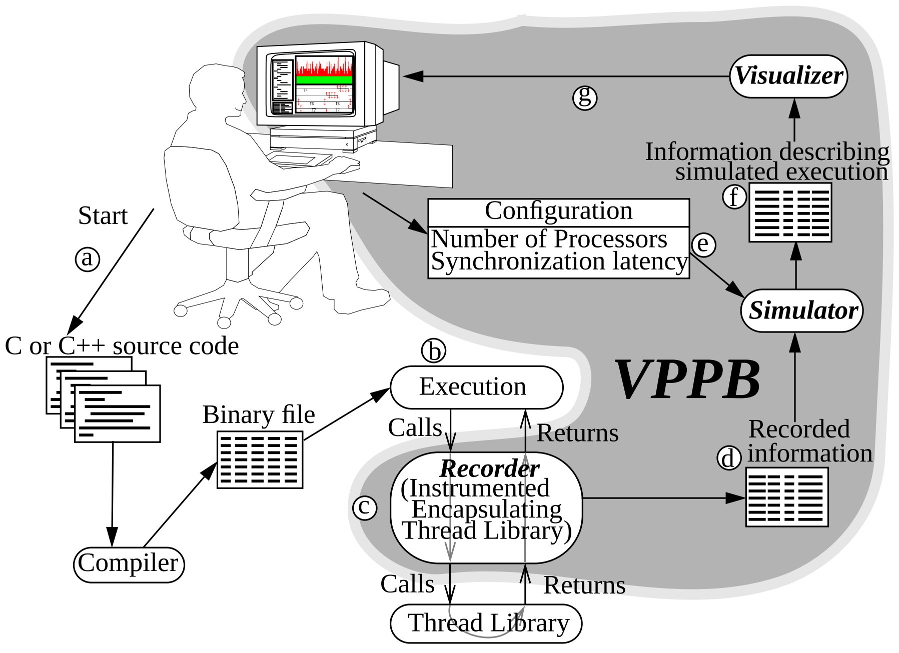


Figure 15: A schematic flowchart of the VPPB system.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 15. The simulator also takes the configuration (e) as input, such as the target number of processors, etc. The output from the simulator is information describing the predicted execution (f). In the simulator we can count the number of threads in the program, the parameter n. We can also count the number of synchronizations in the recorded information, as well as accumulate all execution times recorded in order to get the execution time when using one processor. The later is used together with the number of synchronizations to get the granularity, parameter z. Also the parallel profile vector, parameter V, can be obtained using the simulator. The simulator simulates the recorded information using as many processors as there are threads, then the simulator is able to calculate the parallel profile vector. The simulator was already able to simulate more loosely coupled systems, thus, to set a latency time for the synchronizations between processors [3]. This functionality can be used when comparing a real allocation (predicted by the simulator) with the values given from the function p.

Using the *Visualizer* the predicted parallel execution of the program can be inspected (g). The Visualizer uses the simulated execution (f) as input. The main view of the (predicted) execution is a Gantt diagram. When visualizing a simulation, it is possible for the developer to use the mouse to click on a certain interesting event, get the source code displayed, and the line making the call that generated the event highlighted. The result from the algorithm is shown as a vertical bar, indicating that this is longest time the program has to run if appropriately allocated. With these facilities the developer may detect problems in the program and can modify the source code (a) or change the allocation. Then the developer can re-run the execution to inspect the performance change.

The VPPB system is designed to work for C or C++ programs that uses the built-in thread package [9] and/or POSIX threads [6] on the Solaris 2.X operating system.

7.2. Example - prime number generator

We will demonstrate the described technique (including the tool described in Section 7.1) by bounding the speed-up of a parallel C-program (with Solaris threads) for generating all prime numbers smaller than or equal to a number X. The algorithm generates the primes numbers in increasing order starting with prime number 2 and ending with the largest prime number smaller than or equal to X. A number Y is a prime number if it is not divisible by any prime number smaller than Y. Such divisibility tests are carried out by filter threads; each filter thread has a prime number assigned to it. This is shown in Figure 16. In the front (the first process) there is a number generator feeding the chain with numbers. When a filter receives a number it will test it for divisibility with the filter's prime number. If the number is divisible it will be thrown away. If the number is not divisible it will be sent to the next filter. In the case that there is no next filter, a new filter is created with the number as the new filter's prime number.

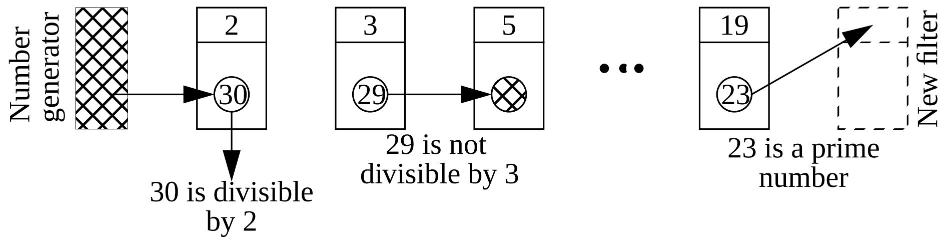


Figure 16: The algorithm for generating prime number.

The demonstration was performed with a number generator that stops by generating number 397, which is the 78th prime number. This means the program will contain 79 threads (including the number generator). The VPPB tool uses the thread programming model which means that the target environment is a (distributed) shared memory machine. The program's speed-up has been evaluated using three kinds of networks; *fast* network with no extra time for a (remote) synchronization, *ordinary* network with a synchronization time of four microseconds, and a *slow* network with eight microseconds latency. The value of the ordinary network has been defined by the remote memory latency found in three distributed shared memory machines: SGI's Origin 2000 [10], Sequent's NUMA-Q [14], and Sun's WildFire [8]. The remote memory latency varies between 1.3 to 2.5 micro seconds [21] and a synchronization must at least include two memory accesses (the thread releasing the thread issues a write and the receiving threads issues a read). The slow network is simply twice the ordinary network. The results are compared to the predictions of the tool in Section 7.1 when applying the corresponding cost for synchronization. The allocation algorithm used in the simulated case is simple. The first n/k processes (process one is the number generator, process two is the first filter, and so on) are allocated to the first processor, the next n/k processes to the second processor and so on. In the case when n is not divisible with k, the first n modulus k processors are allocated $\lceil n/k \rceil$ processes and the remaining processors are allocated $\lfloor n/k \rfloor$ processors. This is a common way to allocate chain-structured programs [4].

As can be seen in Figure 17 (zero latency) the simple allocation scheme performs worse than function p, and we can thus conclude that we for sure are able to find a better allocation. In Figure 18 (latency is four micro seconds) the simple allocation scheme performs worse than function p for less than eight processors, and we can thus conclude that we for sure are able to find a better allocation when we have less than eight processors. In the case with eight processors or more we can not conclude if there is a better allocation or not. However, it is important to know that the fact that we are above the bound does not guarantee that the simple allocation is the optimal. In Figure 19 (latency is eight micro seconds) the simple allocation scheme performs worse than function *p* for four (or less) processors, and we can thus conclude that we for sure are able to find a better allocation when we have less than five processors. In the case with five processors or more we can not conclude if there is a better allocation or not.

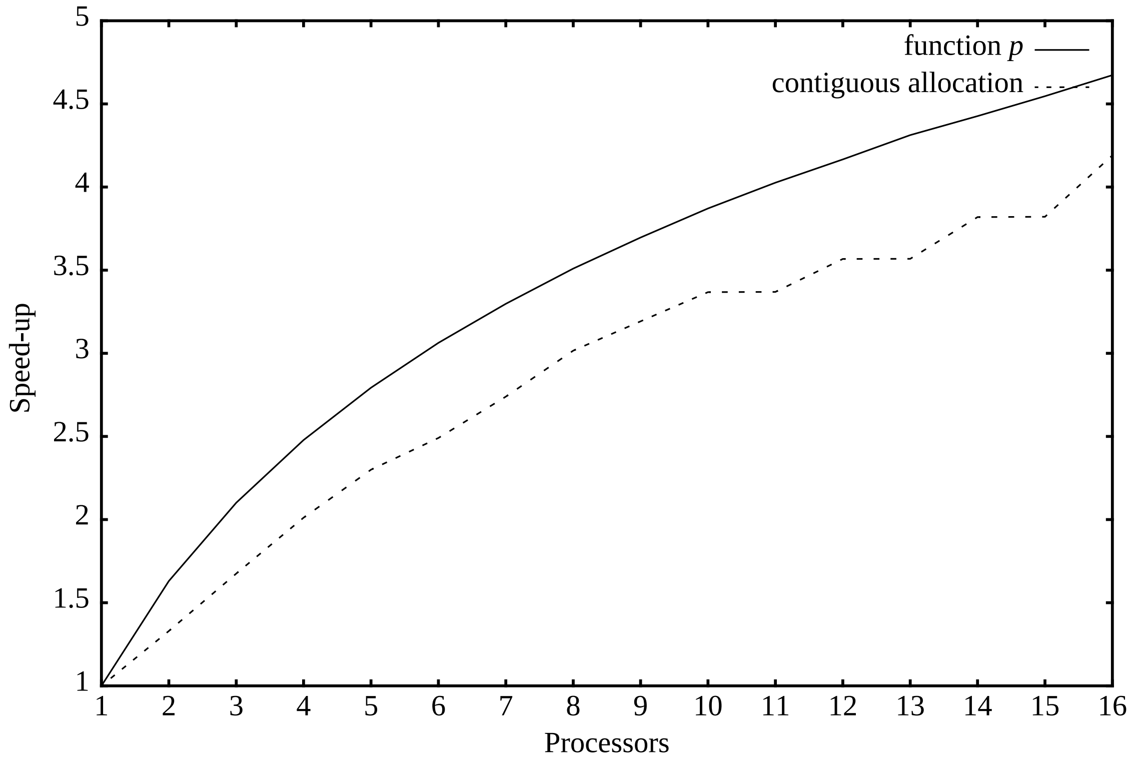


Figure 17: The speed-up for the prime number program on a fast network (latency = 0 μs).

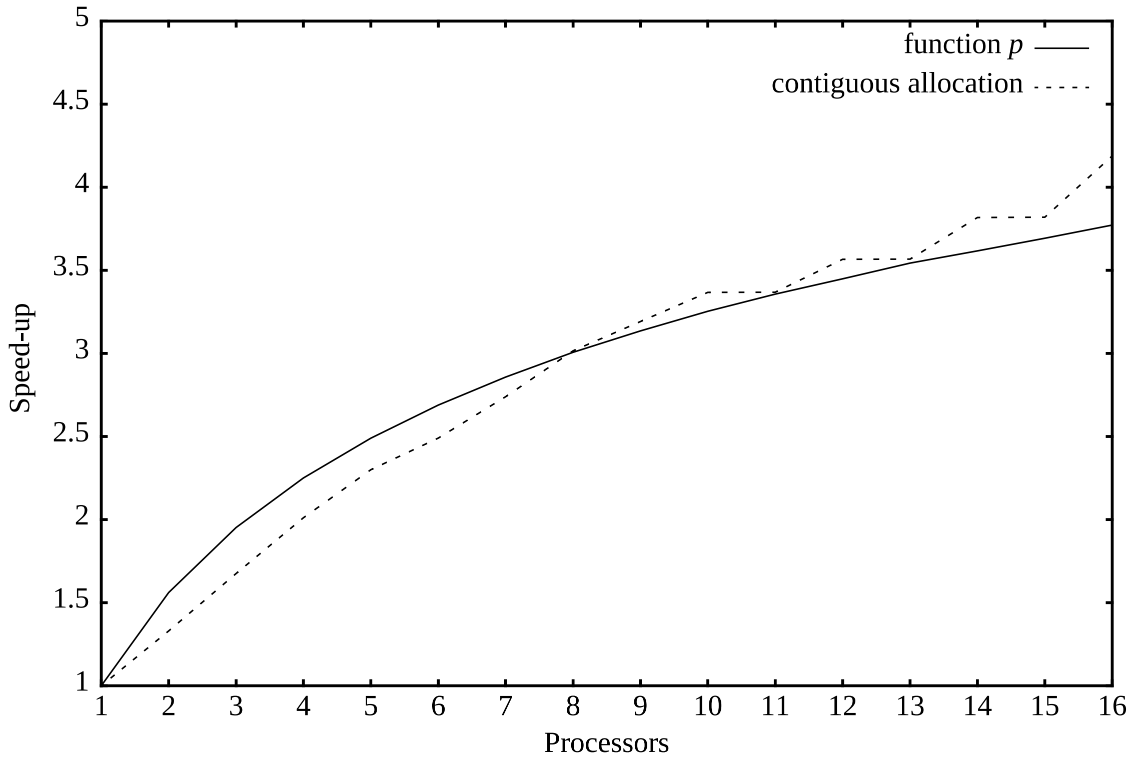


Figure 18: The speed-up for the prime number program on an ordinary network (latency = 4 μs).

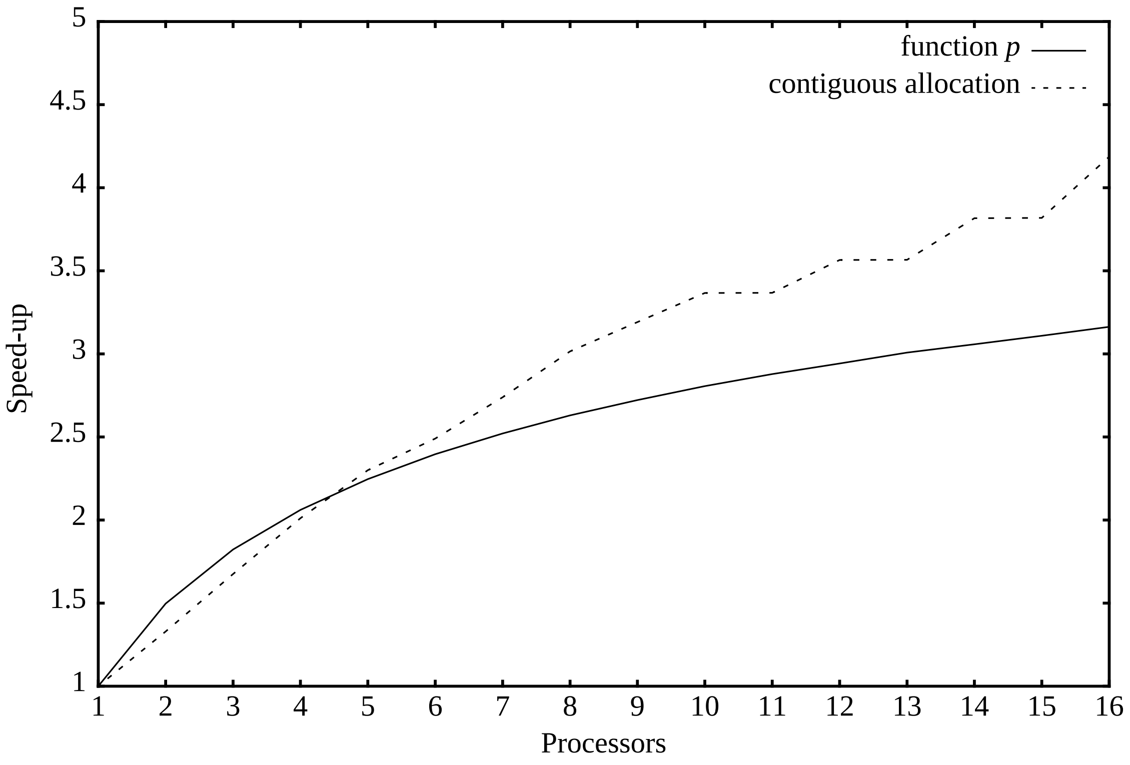


Figure 19: The speed-up for the prime number program on a slow network (latency = 8 μ s).

If we keep the number of processors constant, say at eight, we see how the functions and allocations are affected by the network latency time, shown in Figure 20. The speed-up will decrease as the latency increases. This is clearly shown for function p . The simple allocation is latency tolerant and the speed-up decreases, although hard to see. This is due to allocating the first processes (which also communicated the most) to the same processor, and so on.

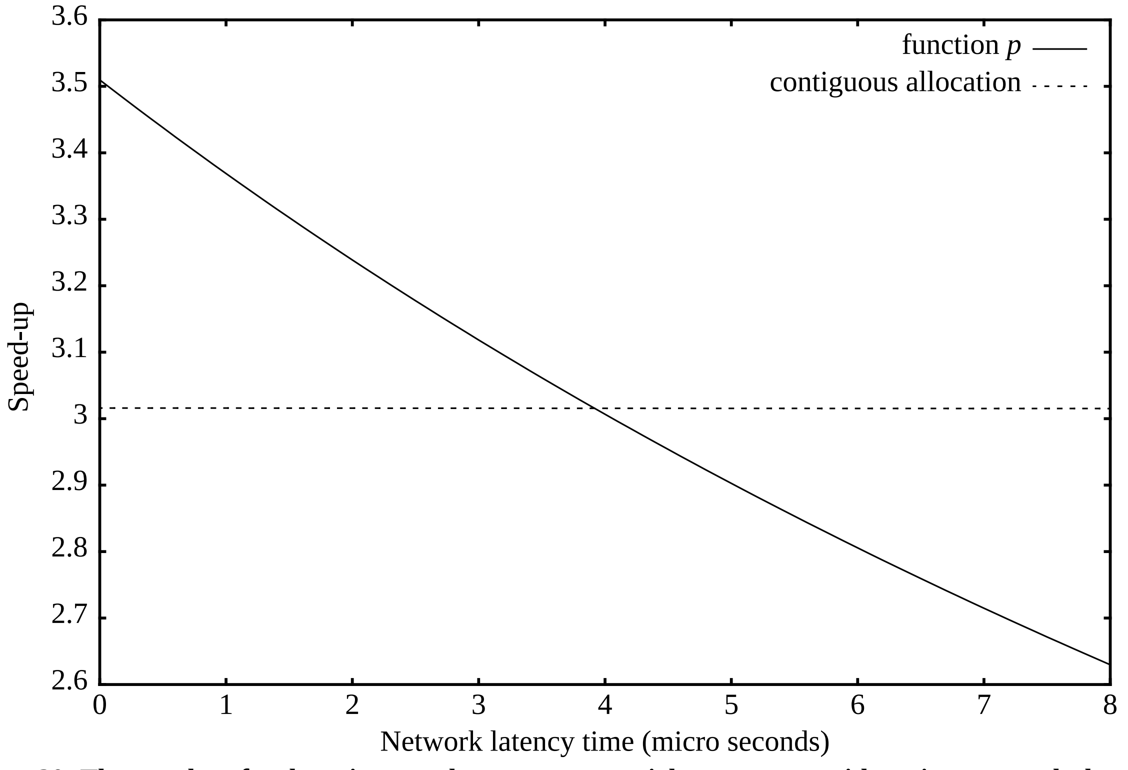


Figure 20: The speed-up for the prime number program on eight processors with various networks latencies.

As indicated earlier, the function *p* gives a bound. The intended usage of this bound is to indicate that there is a better allocation strategy if the current allocation strategy gives results below the bound, as in the case in Figure 17 (the prime program with no latency). We now look at other allocation strategies and tries a round robin strategy that assigns (on a multiprocessor with *k* processors) thread number *i* to processor (*i* mod *k*)+1. In Figure 21 we see that the round robin strategy gives better result than the bound. However, a better result than the bound does not automatically means that we have found an optimal result.

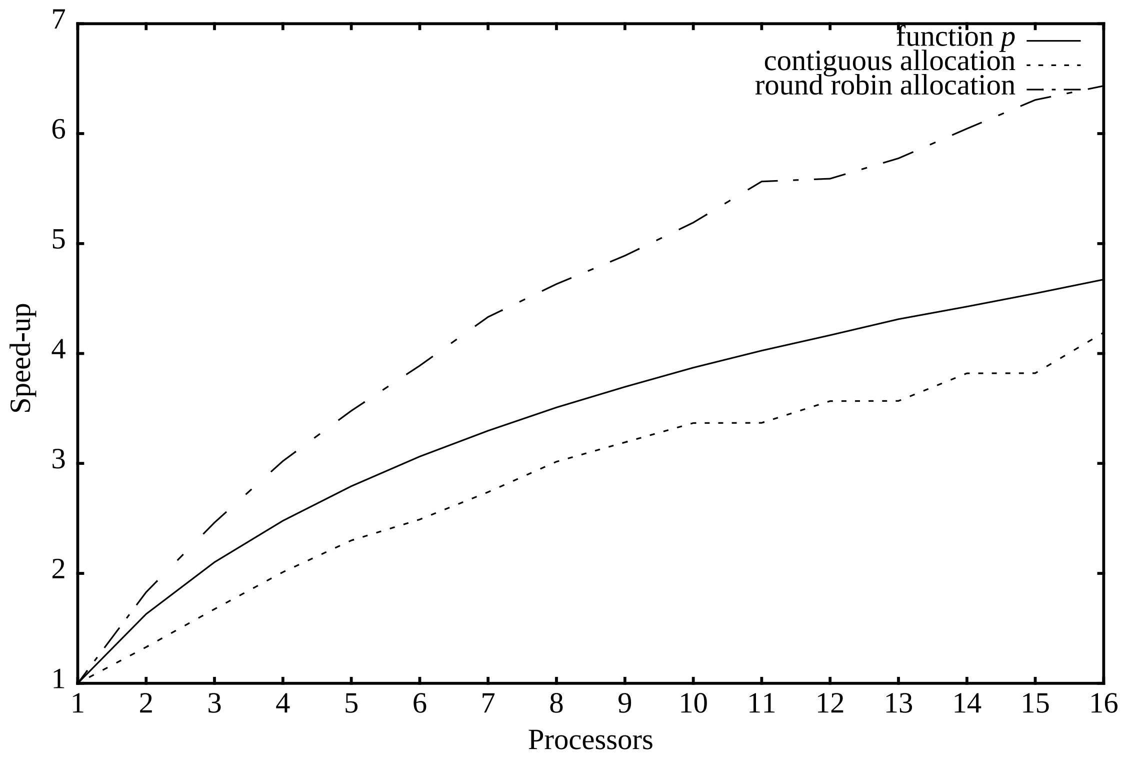


Figure 21: The speed-up using round robin for the prime number program (latency = 0 μ s).

In Figure 22 the previous graphs are summarized. The figure shows the function *p* for a latency between 0 and 8 micro seconds and the number of processors between 1 and 16.

It took in average 102 seconds to calculate a data point in the graph in Figure 22 on a 300 MHz Sun Ultra 10 workstation. The reason for finding the optimal solution so fast is the use of allocation classes and the branch-and-bound algorithm combined with the lazy evaluation, described in Section 6. One might think that if we desire to find the optimal solution, all the calculations with allocation classes will be waisted since they are all (except the last) unrealistic allocations (different allocations for the thin and thick part). On the contrary, by traversing the tree from the top we are able very efficiently ignore several branches. The reason for this is that an allocation class will always have worse (or equal) results in its sub-classes. We are thus able reduce the search dramatically. We are able to reduce some of the calculation costs as well. The major calculation cost is for the thick part and in Figure 13(d) we can see that the top allocation class has the same thick part allocation as the previous allocation class in Figure 13(c). Then we can re-use the already calculated value of the thick part, however the two remaining allocations in Figure 13(d) have to be calculated. Note that we always calculate the thin part, since it is very fast to calculate.

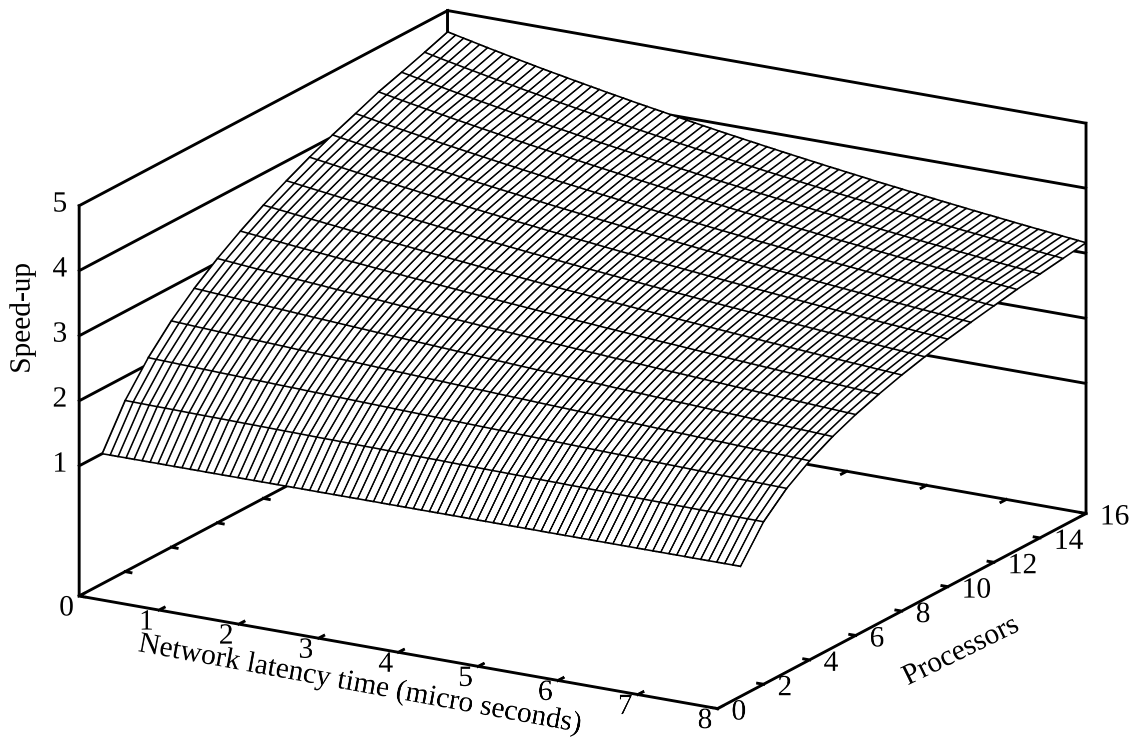


Figure 22: The speed-up for the prime number program by applying function *p*, varying number of processors and networks latencies.

In order to illustrate the benefits of our algorithm we consider the prime number program on 16 processors. In total we can find 6,158,681 unique allocations of the 79 processes on 16 processors. The number of allocations that needs to be evaluated using our approach is very small, in Figure 23 the fraction of allocations calculated using branch-and-bound compared to extensive search is shown. As can be seen, roughly only one allocation out of 80,000 needs to be evaluated even without the lazy evaluation. If we add the lazy evaluation we have to evaluate roughly one allocation out of 270,000. The time to calculate the optimal allocation for the prime number program ($k = 16, t = 8\mu s$) is five minutes on a 300MHz Ultra 10. Without the allocation classes, branch-and-bound technique and lazy evaluation the same calculation would require two and a half year.

8. Discussion and Related Work

The parameters used by the method do not require such an advanced tool as the one used in Section 7. However, in order to get the programs parameters we still need some kind of tool. In order to calculate the granularity, z, we need the programs total execution time (on one processor) which can be obtained by executing the program on a single processor workstation and some time-command to measure the time. The number of synchronizations must, however, be measured in some other way. One way could be to use the (unsupported) tool from Sun called tnfview [22] which graphically shows all threads and synchronizations. Although possible to calculate the number of synchronizations by hand a short script would do the work. The tnfview also shows the number of threads, i.e. the parameter n.

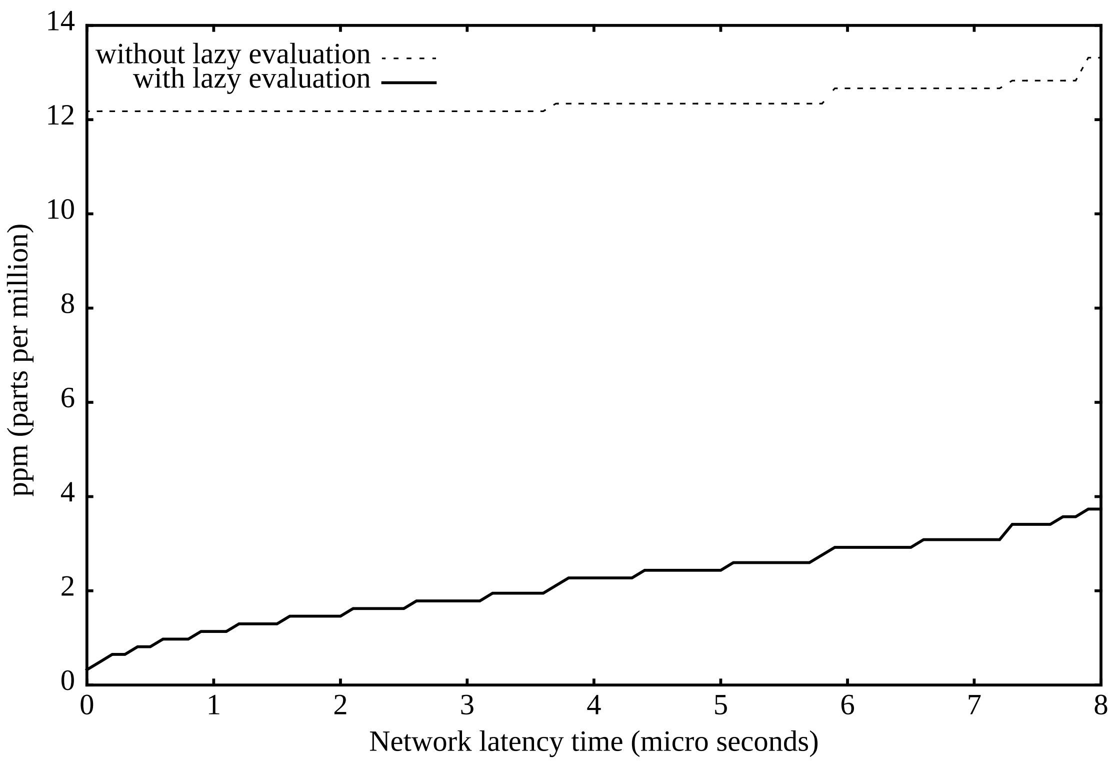


Figure 23: The fraction of allocations calculated using allocation classes and branch-and-bound with and without the lazy evaluation compared to extensive search, expressed in ppm (parts per million).

The remaining parameter, the parallel profile vector V, is somewhat harder to obtain than n and z. Here we need to find the amount of parallelism in the program when there are one processor for each process. For many systems the number of processors in a multiprocessor is less than the number of threads in the program. We can use tnfview (with some limitations as discussed and solved for our tool in [2]) in order to capture the synchronizations between threads and calculate the sequential work time between the synchronizations. With this information we can build a weighted directed graph where each node is a sequential segment (with the work time as weight) and where an edge is a synchronization. With this graph we can obtain the parallel profile vector V. The length of the longest path in the graph will represent $T(P, n, 0)$.

Further development of the method could be to adjust the formulas to include that several processors may be situated on the same node, sharing the same physical memory, thus the latency is then between processors on different nodes and not between processor on the same node. Processes may also be dynamically scheduled from one processor to another within the same node.

The basic approach in the current work is to obtain a performance bound, which we know is possible to achieve. This approach has been used for a long time in real time systems, where it is well known that a set of n tasks is schedulable if the sum of processor utilization is less than $n(2^{(1/n)} - 1)$ [5], similar results also exist for real-time process sets which operates under what is called an age constraint [19][20].

The proof techniques described in Section 4 in this paper has previously been used for obtaining performance bounds in a number of application areas besides multiprocessor scheduling, e.g. cache memory systems [13], parallel accesses to multiprocessor memory systems [16], and message scheduling on a number of communication links [15].

9. Conclusion

Cluster systems do not usually allow processes to dynamically migrate from one node to another. It is thus essential to find an efficient allocation of processes to processors. Even in the case of machines with distributed shared memory such allocations can be beneficial. Unfortunately, finding the optimal allocation is NP-hard and we have to resort to heuristic algorithms to find a good allocation. One problem with heuristic algorithms is that we do not know whether the result is close to optimal or if it is worth-while to continue the search for a better result.

In this paper we find a way to analytically calculate a bound on the minimal completion time for a given parallel program. The program is specified by the parameters n (the number of processes), V (the parallel profile vector), and z (the synchronization frequency); different programs may yield the same n, V, and z. Further, the hardware is specified by the parameters k (number of processors) and t (the synchronization latency). The bound on the minimal completion time is optimally tight. This means that there exists a program (with the given parameters n, V, and z) for which the minimal completion time is equal to the bound. The parameters z and t give a more realistic model than previous work [17][18]. The bound makes it possible for the user to determine when it is worth-while to continue the heuristic search, or perhaps apply another heuristic algorithm. Analytical performance bounds, like the one presented here, have successfully been used when engineering real-time systems, and techniques similar to the one described here has been used for network and memory systems. Our method includes an aggressive branch-and-bound algorithm that, together with lazy evaluation, has been shown to reduce the search space to only 0.0004% for finding the optimal bound.

The method has been implemented in a practical tool. The tool is able to automatically obtain the values of n, V, and z for a program and calculate the minimal completion bound with the given hardware parameters. Based on this practical tool we have demonstrated the usability of the method. One feature of the tool is that it can be totally operated on a single processor workstation. This is of practical importance since no multiprocessor is then needed when applying the method. Finally, the tool makes the presented analytical bound on the minimal completion time easily accessible for practitioners.

Acknowledgment

This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

References

[1] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, “Scheduling Computer and Manufacturing Processes,” Springer-Verlag, ISBN 3-540-61496-6, 1996.

[2] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads,” in Proceedings of the 13th International Parallel Processing Symposium, pp. 407-413, 1999.

[3] M. Broberg, L. Lundberg, and H. Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs,” in Proceedings of the 12th International Parallel Processing Symposium, pp. 770-776, 1998.

[4] S. H. Bokhari, “Partitioning Problems in Parallel, Pipelined, and Distributed Computing,” IEEE transactions on Computers, Vol. 37, No. 1, 1988.

[5] A. Burns and A. Wellings, “Real-Time Systems and Programming Languages,” Addison-Wesley, 2001, ISBN 0-201-72988-1.

[6] D. Butenhof, “Programming with POSIX Threads,” Addison-Wesley, ISBN 0-20-163392-2, 1997.

[7] M. Garey and D. Johnson, “Computers and Intractability,” W. H. Freeman and Company, 1979.

[8] E. Hagersten, and M. Koster, “WildFire: A Scalable Path for SMPs,” in Proceedings of the The Fifth International Symposium on High Performance Computer Architecture, pp. 172-181, 1999.

[9] S. Kleiman, D. Shah, and B. Smaalders, “Programming with threads,” Prentice Hall, 1996.

[10] J. Laudon and D. Lenoski, “The SGI Origin: A ccNUMA Highly Scalable Server,” in Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 241-251, 1997.

[11] H. Lennerstad and L. Lundberg, “An optimal Execution Time Estimate of Static versus Dynamic Allocation in Multiprocessor Systems,” SIAM Journal of Computing, Vol 24, no 4, pp. 751-764, 1995.

[12] H. Lennerstad and L. Lundberg, “Optimal Combinatorial Functions Comparing Multiprocess Allocation Performance in Multiprocessor Systems,” SIAM Journal of Computing, Vol. 29, No. 6, pp. 1816-1838, 2000.

[13] H. Lennerstad and L. Lundberg, “Optimal Worst Case Formulas Comparing Cache Memory Associativity,” SIAM Journal of Computing, Vol. 30, No. 3, pp. 872-905, 2000.

[14] T. Lovett and R. Clapp, “STiNG: A cc-NUMA Computer System for the Commercial Marketplace,” ACM/IEEE International Symposium on Computer Architecture (ISCA), pp. 308-317, 1996.

[15] L. Lundberg, H. Lennerstad, “Optimal Bound on the Gain of Permitting Dynamical Allocation of Communication Channels in Distributed Processing,” Acta Informatica, Vol. 36, No. 6, pp. 425-446, 1999.

[16] L. Lundberg, H. Lennerstad, “Bounding the Gain of Changing the Number of Memory Modules in Shared Memory Multiprocessor,” Nordic Journal of Computing, Vol. 4, No. 3, 1997, pp. 233-258.

[17] L. Lundberg, H. Lennerstad, “Using Recorded Values for Bounding the Minimum Completion Time in Multiprocessors,” IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 4, pp. 346-358, 1998.

[18] L. Lundberg, H. Lennerstad, “An Optimal Upper Bound on the Minimal Completion Time in Distributed Supercomputing,” in Proceedings of the 1994 ACM International Conference on Supercomputing, pp. 196-203, 1994.

[19] L. Lundberg, “Fixed Priority Scheduling of Age Constraint Processes,” in Proceedings of the 4th International Euro-Par Conference, pp. 288-296, 1998.

[20] L. Lundberg, “Utilization Based Schedulability Bounds for Age Constraint Process Sets in Real-Time Systems,” Real-Time Systems, to appear.

[21] L. Noordergraaf and R. van der Pas, “Performance Experiences on Sun’s WildFire Prototype,” in Proceedings of the Super Computing Conference 1999, CD-ROM, 1999.

[22] SunSoft, “tnfview Demo Guide,” 1995. The documentation is included in ftp://ftp.sunsite.kth.se/ftp/www/sun/sunsite-sun-info/tnftools/tnfdemo.tar.gz, (site visited November 20th, 2001).

A Method for Bounding the Minimal Completion Time in Multiprocessors
by Jan Magnus Broberg, Lars Lundberg, Kamilla Klonowska

ISSN 1103-1581
ISRN BTH-RES--03/02--SE