University of Edinburgh

Department of Computer Science

Topology Effects on Scheduling in DM-MIMD Machines

MSc Project

Michał Cierniak

September, 1991

Abstract

Scheduling programs represented by task graphs (dags) onto multiprocessors represented by processor graphs (undirected graphs) is tackled in this project. A new linear time heuristic (MMH) is proposed for task scheduling. MMH is a simplified version of the Mapping Heuristic. Execution times of programs scheduled by MMH onto different multiprocessors are compared. Finally, a simple way of estimating a quality of a topology is proposed.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the following people:

My supervisor, Damal Arvind for help and encouragement throughout the course of my project.

My mentor, Stephen Gilmore, and my fellow students for enduring my talks about the project.

Sathiamoorthy Manoharan for listening to my questions concerning the depth-first breadth-next heuristic and scheduling in general.

All the friendly souls for advice how to use Unix tools.

Chapter 1

Introduction

1.1 Motivation

The role of parallel computers will be significant in the near future. One way of speeding programs within the same hardware technology is to execute them on multiprocessors. One of the biggest challenges facing compiler writers is to efficiently implement programming languages on parallel computers.

There are three fundamental problems to be solved when compiling a program for parallel execution on a multiprocessor:

1. Identifying potential parallelism.
2. Partitioning the program into sequential tasks.
3. Scheduling the concurrent execution of these tasks.

Only the last problem is addressed in this project. Solving the first two problems is relatively straightforward and can be done efficiently if a suitable programming language is chosen ([Sar89]). It can be done, however, even for languages originally designed for sequential computers ([BJP91]).

There are two cases of scheduling the execution of the tasks. In the more general one, a partial ordering is imposed on the tasks, and so two problems must be solved:

- Each task must be assigned to one of the processors of the parallel computer.
- For tasks assigned to the same processor the best order of their execution must be found.

In the second, simpler case, all tasks are assumed to exist during the whole execution time of the program. The only problem to be solved is then to assign tasks to processors in such a way that the computational load is distributed evenly between all the processors with minimal communication.

The former problem is called *scheduling* and the latter — *mapping*. Unfortunately both problems are known to be *NP*-complete. This fact makes finding optimal solutions infeasible. Because of their importance both mapping and scheduling have been extensively researched by many scientists and considerable literature exists on this subject.

In the initial stage of this project both mapping and scheduling were investigated. After the literature survey it was decided that scheduling was more interesting — it was more general with scope for design of fast heuristics.

Until recently most of the work in this area assumed simplified models of the programs to be scheduled and the target computer architectures. The less realistic of the simplifying assumptions were the following:

1. The number of processors is unbounded.
2. All the tasks are independent ([HS88], [GIS77], [BDW86]).
3. There are no message transfers between tasks ([ACD74]).
4. There is a link between any pair of processors ([Chr89], [Dar91]).

For many applications, however, these assumptions are invalid. Real computers contain a bounded number of processors. Messages of different sizes are sent

between pairs of tasks. Processors in computers are connected in some patterns, not with links between all pairs of processors. These facts are reflected in the more recent work ([ERL90], [MT91], [HCAL89]).

The scheduling algorithm implemented in this project, like most of the existing algorithms, is *static* — the task execution times and message sizes are estimated at compile-time. Such algorithms have the advantage of low run-time overhead and scope for finding globally optimal schedules. The obvious disadvantage is that in some cases the true task execution times can be estimated only at run-time.

The scheduling algorithm implemented in this project belongs to a class called *list schedulers*. In list scheduling, each task is assigned a priority. Whenever a processor is available, a task with the highest priority is selected from the list of ready tasks and assigned to a processor. The schedulers in this class differ only in the way that each scheduler assigns priorities to tasks. The algorithm implemented in this project is a modification of the *mapping heuristics* (**MH**) described in [ERL90], which in turn is an improved list scheduler using an HLFET (Highest Level First with Estimated Times) scheme of assigning priorities ([ACD74]). The *modified mapping heuristics* (**MMH**), implemented as a part of this project, gives similar solutions as the simpler version of **MH**. It is faster than the full **MH**. The **MMH** time complexity is $O(m(n+e)+m^3)$ as compared to **MH** complexity: $O(m^3e+m(n+e))$, where n is the number of tasks, e — number of direct dependencies between tasks, and m — number of processors.

The full **MH** takes into account contention on the multiprocessor links, which the simpler version of **MH** and the **MMH** disregard. The decision of implementing the simpler version of the scheduler was taken because the **MH** scheduler implemented only one routing algorithm and obviously in different multiprocessors different routing algorithms are used. Thus the results of scheduling are not completely correct anyway and the overhead of taking contention into account is relatively high.

Programs to be scheduled are represented in this project as *task graphs*. Task graphs are *directed acyclic graphs* (dags). Vertices and edges represent tasks and

communication between them. Several values are associated with each task. They define the computation time and hardware requirements (e.g. amount of memory) that must match hardware characteristics of the processor. A value specifying the message size is associated with each edge of the task graph.

Multiprocessors are represented as *processor graphs*. These are undirected graphs since links between processors are assumed to be bidirectional. Each processor is described by its processing speed and hardware characteristics. Links are described by the *link capacity* (communication speed) and the *initialization time*.

A scheduler used in the project for various experiments employs **MMH** to generate a schedule for a given task graph and a processor graph. Then the execution of the program scheduled in this way is simulated. In the simulation the shortest path routing is used and contention is taken into account. The output of the simulator contains the achieved speedup over the sequential execution of the program.

Experiments performed in the course of the project involved running the scheduler for different data sets. Various task graphs were used — both examples from real applications and randomly generated dags. The effects of changing the *granularity* of the task graphs were investigated. The granularity was defined as a ratio of the time required to send all the messages through a single link to the computation time of all tasks on a single processor.

On the other hand various processor graphs were used. The interconnection topology and number of processors were changed. Topologies included *ring*, *mesh*, *torus*, *hypercubes*, *trees*, *de Bruijn*, and *Kautz* graphs.

1.2 Results

The speedups and processor utilizations for different topologies of processor graphs and different granularities of the task graphs were analysed and compared.

Then a way of estimating the "goodness" of a topology was proposed. The

goodness factor was defined as a ratio of the average vertex degree to the average internode distance.

A strong correlation between the goodness factor and speedups achieved for different topologies was discovered. The enumeration of topologies in order from the "best" (allowing for the greatest speedup) to the "worst" is as follows: enhanced hypercube (k=0), Kautz graph - UK$(3, x)$, hypercube, Kautz graph - UK$(2, x)$, binary de Bruijn graph, symmetrical torus, mesh, enhanced binary tree, ring, complete binary tree.

1.3 Thesis overview

This thesis consists of eight chapters. Chapter two contains definitions of some basic concepts used in the rest of the thesis. In chapter three some important multiprocessor topologies are presented. Their definitions are given and the most important properties described. Chapter four includes an overview of mapping — one of the approaches to the problem of distributing parallel programs onto DM-MIMD machines. Existing scheduling algorithms are presented in chapter five. Chapter six describes the Modified Mapping Heuristic and compares it with three existing algorithms. Experiments comparing different topologies are described in chapter seven. The results of the experiments are discussed and the chapter is concluded by proposing a "goodness factor" — a way of estimating a quality of a topology. Chapter eight summarizes the results and gives some directions for further work.

Chapter 2

Basic definitions

2.1 Graphs

Graphs and graph algorithms were basic tools used in the project. Because the nomenclature useed in in graph theory is not consistent — different authors use different names for the same concept and identical names for different concepts — I will introduce here some of the crucial concepts.

A graph $G = \langle V, E \rangle$ is defined by a set V of *vertices* and a set E of *edges*. Number of vertices is called the *order* of a graph. Number of edges is called the *size* of a graph.

Two vertices joined by an edge are said to be *adjacent*. Graphs considered in this project have no *loops* (i.e. edges with one end) and no *multiple edges* — any pair of vertices is joined by at most one edge.

A *degree* of a vertex v is the number of vertices adjacent to v.

A *distance* between two vertices u and v is the minimum of the lengths of paths from u to v.

A *diameter* is the largest distance between two vertices in a graph.

A *clique* (or a *connected graph* is a graph with each pair of vertices being joined by an edge.

2.2 Digraphs

A *digraph* $\Gamma = \langle V, E \rangle$ is defined by a set V of vertices and a set E of *arcs*. Two vertices are associated with each arc, the *head* and the *tail*.

The *in-degree* of a vertex $v \in V$ is the number of arcs of Γ having v as head. The *out-degree* of v is the number of arcs having v as tail.

A digraph is *n-regular*, if and only if, each vertex has in-degree and out-degree equal to n.

An *undirected acyclic graph* (dag) is a digraph with no cycles.

A *source* is a vertex with in-degree equal 0. A *sink* is a vertex with out-degree equal 0.

Chapter 3

Graph topologies

Different multiprocessor interconnection patterns were investigated in this project. Some of the graphs are better suited as multiprocessor interconnection networks. The results showing speedups achieved for different processor graphs are given in Chapter 7 of this report. In this chapter, the definitions and basic properties of the following graphs (which are used as processor graphs) are given: ring, mesh, torus, complete binary tree, hypercube, enhanced hypercube.

3.1 Ring

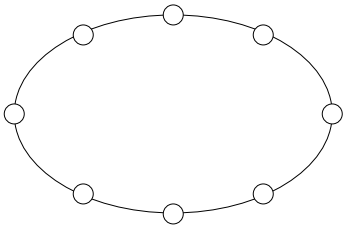


Figure 3.1: A ring

The ring has a desirable property of the degree of all vertices being constant and equal to 2 (a ring is 2-regular). The diameter is unfortunately, $D = \lfloor \frac{|V|}{2} \rfloor$. For $|V| > 2$ the size is $|E| = |V|$.

3.2 Mesh

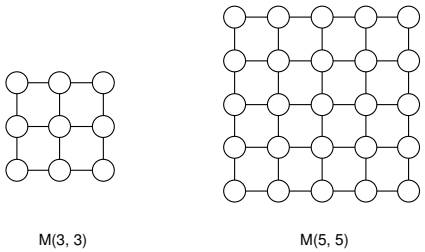


Figure 3.2: Meshes

In this project only two dimensional, square meshes are considered. They are called here symmetrical meshes. A symbol used to denote a mesh with width x and height y is $\mathrm{M}(x, y)$. A symmetrical mesh $\mathrm{M}(n, n)$ will be denoted as $\mathrm{MS}(n)$. The maximum vertex degree of a mesh is 4, and the diameter, $D = 2(\sqrt{|V|} - 1)$. The size is, $|E| = 2(|V| - \sqrt{|V|})$.

3.3 Torus

In practice, all the processors in one computer have the same number of links. Therefore, it is uneconomical to have some of the vertices with a degree less than the maximum vertex degree. We can augment the mesh topology so that all vertices have the degree, $\Delta = 4$. A graph like this may be viewed as a torus.

A torus obtained in this way from a mesh $\mathrm{M}(x, y)$ will be denoted $\mathrm{T}(x, y)$. Again, $\mathrm{TS}(n)$ is an abbreviation for $\mathrm{T}(n, n)$. The diameter of a symmetrical torus is, $D = 2\lfloor\frac{\sqrt{|V|}}{2}\rfloor$. The size, $|E| = 2|V|$.

3.4 Complete binary tree

The order of $\mathrm{CBT}(n)$ is $|V| = 2^n - 1$. Each of the vertices v_i, $i = 1, 2, \ldots, 2^{n-1} - 1$, has two descendants: v_{2i} and v_{2i+1}. Each of the vertices v_i, $i = 2, 3, \ldots, 2^n - 1$, has a parent: $v_{\lfloor\frac{i}{2}\rfloor}$. Therefore the maximum degree is $\Delta = 3$. The size of $\mathrm{CBT}(n)$ is $|E| = 2^n - 2$. The diameter is $D = 2(n - 1) = 2(\log_2(|V| + 1) - 1)$.

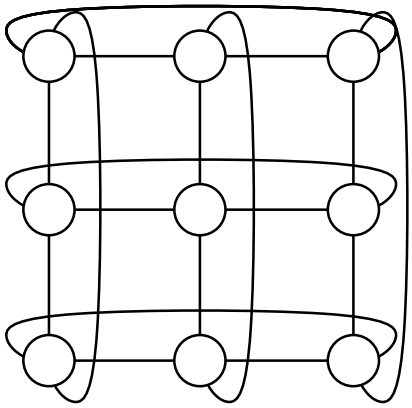


Figure 3.3: A symmetrical torus

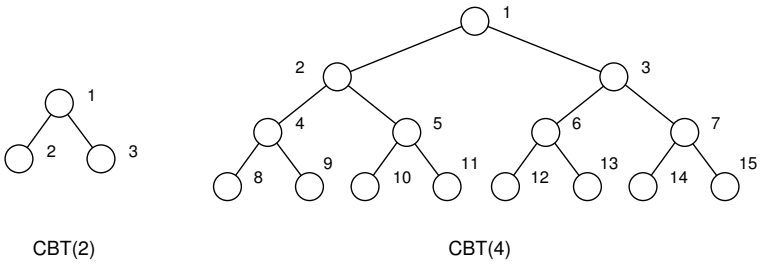


Figure 3.4: Complete binary trees

3.5 Hypercube

A hypercube is one of the topologies widely used for interconnecting processors in parallel computers. Examples of multiprocessors with this interconnecting pattern include: iPSC ([Sco91a]) and NCUBE ([Sco91b]).

Definition 3.5.1 *A hypercube* HC(n) *(also called* n*-cube) is an undirected graph consisting of* $k = 2^n$ *vertices labeled from* 0 *to* $2^n - 1$ *and such that there is an edge between any two vertices, if and only if, the binary representation of their labels differ by one bit.*

□.

Saad and Schultz discuss in [SS88] many properties of hypercubes. The most fundamental of these are:

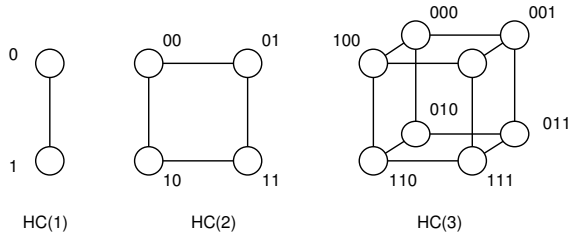


Figure 3.5: Hypercubes

- HC(n) can be constructed recursively from two cubes HC($n-1$) by joining every vertex of the first $(n-1)$-cube to the vertex of the second having the same label.
- There are n different ways of tearing an n-cube, i.e. splitting it into two $(n-1)$-subcubes so that their respective vertices are connected in a one-to-one way.
- Any two adjacent vertices A and B of an n-cube are such that the nodes adjacent to A and those adjacent to B are connected in a one-to-one fashion.
- All cycles in a hypercube have even lengths.
- The n-cube is a connected graph of diameter n.

The properties that can be used to recognize hypercubes are stated in the following theorem:

A graph $G(V, E)$ is an n-cube if and only if

1. V has 2^n vertices;
2. every vertex has degree n;
3. G is connected;
4. any two adjacent vertices A and B are such that the vertices adjacent to A and those adjacent to B are linked in a one-to-one fashion.

Distances and Paths in Hypercubes

It is obvious from the definitions that the maximum distance between any two vertices A and B is equal to number of bits that differ between A and B, i.e. to the Hamming distance $\mathrm{H}(A, B)$.

In many applications it is important to find several vertex-disjoint paths (*parallel paths*) between two vertices. Two properties of a hypercube are given:

- There are $\mathrm{H}(A, B)$ parallel paths of length $\mathrm{H}(A, B)$ between any two nodes of a hypercube.
- There are n parallel paths of length at most $\mathrm{H}(A, B) + 2$ between any two nodes of an n-cube.

Mapping other Geometries into Hypercubes

The problem of mapping a topology onto a physical interconnection network is very common in the use of multiprocessors. Very often an effective algorithm for a given geometry exists and we would like to execute it on a general purpose multiprocessor. The properties of a hypercube are very good in this respect. It is possible to map ring, linear arrays, grids of arbitrary dimensionality, and trees onto a hypercube.

The use of Gray codes for mapping rings and grids into hypercubes is described in detail in this article.

The size of a $\mathrm{HC}(n)$ is $|E| = n2^{n-1}$. The diameter is, $D = n = \log_2 |V|$ and the maximum degree is $\Delta = n = \log_2 |V|$ ($\mathrm{HC}(n)$ is n-regular).

3.6 Enhanced hypercube

There are several ways af enhancing a hypercube, i.e. adding edges to improve some of the measures of a hypercube (e.g. diameter, mean distance between vertices,

traffic density over a link, vertex degree, or routing control complexity). One of methods of enhancing a hypercube was proposed by Tzeng and Wei in [TW91b].

Let a binary label of a vertex of an n-cube introduced in Definition 3.5.1 be $(x_{n-1} \cdots x_1 x_0)$.

An edge is added between every pair of nodes, $(x_{n-1} \cdots x_{n-k} x_{n-k-1} \cdots x_i \cdots x_0)$ and $(x_{n-1} \cdots x_{n-k} \bar{x}_{n-k-1} \cdots \bar{x}_i \cdots \bar{x}_0)$, with $0 \leq k \leq n-2$.

The three dimensional hypercubes with $k = 1$ and $k = 0$ are sketched in Figure 3.6.

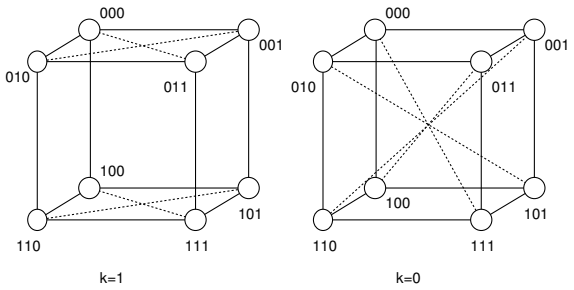


Figure 3.6: Enhanced hypercubes

Tzeng and Wei in [TW91b] give and prove correctness of the shortest path routing algorithm. They also give a formula for the diameter, $D = k + \lceil (n-k)/2 \rceil$.

The optimal values of k with respect to various performance measures are given:

- With respect to mean internode distance (for uniform traffic distributions), $k_{\text{opt}} = 0$
- With respect to diameter, $k_{\text{opt}} = 0$.
- With respect to traffic density (for low reference locality), $k_{\text{opt}} = 0$ for even n, and $k_{\text{opt}} = 1$ for odd n.

The article [TW91b] is concluded with a broadcast algorithm and some pragmatic considerations. The latter part includes a criterion for comparing different structures. It is defined as $(\Delta \times D)$, where Δ is a degree of vertices and D is a diameter. The authors argue that this value is a good criterion for comparing

different topologies. The measure of $(\Delta \times D)$ is better in enhanced hypercubes than in regular hypercubes.

Enhanced hypercubes with $k = 0$

In this project only enhanced hypercubes with $k = 0$ (which is optimal with respect to most measures) were considered.

The symbol EHC(n) will be used to describe an enhanced hypercube with $k = 0$ based on HC(n)

The diameter of EHC(n) is $D = \lceil \frac{n}{2} \rceil = \lceil \frac{\log_2 |V|}{2} \rceil$, the degree $\Delta = n + 1 = \log_2 |V| + 1$, the size $|E| = (n+1)2^{n-1}$, the order is of course unchanged with respect to a regular hypercube and is equal to 2^n.

3.7 De Bruijn graphs

De Bruijn digraphs

The most natural way to define de Bruijn graphs (and Kautz graphs, cf. Section 3.8) is as digraphs. The approach is similar to the way of defining a hypercube (cf. Definition 3.5.1). This definition is given after [BP89].

Definition 3.7.1 *The de Bruijn digraph* B(d, D) *is the digraph whose vertices are the words of length D on an alphabet of D symbols (or d-ary D-tuples). There is an arc from a vertex $\boldsymbol{x}$ to a vertex $\boldsymbol{y}$ if the $D-1$ first symbols of $\boldsymbol{y}$ are equal to the $D-1$ last symbols of $\boldsymbol{x}$. That is, there is an arc from $(x_D x_{D-1} \cdots x_1)$ to all the vertices $(x_{D-1} x_{D-2} \cdots x_1 \alpha)$ where α is any symbol of the alphabet (shifting property).*

□.

Therefore B(d, D) is d-regular, its diameter is D and its order is $|V| = d^D$.

There exist also other definitions of de Bruijn digraphs ([BP89]), e.g.:

- a definition using line digraph iterations
- a definition from congruences

A de Bruijn digraph B(d, D) has the interesting property that there exists exactly one path of length D between any two vertices.

Undirected de Bruijn graphs

Definition 3.7.2 *The undirected de Bruijn graph* UB(d, D) *is the undirected graph obtained from the* B(d, D) *by forgetting orientations of the arcs, removing self loops, and replacing each double edge by a single edge.*

□.

Note: There is an alternative definition of the undirected de Bruijn graph ([SP89]) which preserves self loops and double edges. In this project only the Definition 3.7.2 is used.

Now the vertex $(x_D x_{D-1} \cdots x_1)$ is adjacent (or equal) to all vertices $(x_{D-1} \cdots x_1 \alpha)$ and $(\alpha x_D \cdots x_2)$, where α is any symbol in the alphabet. Hence the maximum degree Δ is equal to $2d$ (note that UB(d, D) is not regular). The order is $|V| = d^D$.

It is interesting to note that de Bruijn graphs use edges more "economically" than hypercubes. E.g. a hypercube with diameter and maximum degree equal to 10 has 1024 vertices, whereas de Bruijn graph with the same properties has 9,765,625 vertices and Kautz graph (cf. Section 3.8) 11,718,750. Or, to find another example, a hypercube with 256 vertices has a diameter $D = 8$ and maximum degree $\Delta = 8$, de Bruijn graph with 256 vertices and diameter equal to 8 has the maximum degree of only 4.

De Bruijn graphs are promising and they have not been investigated as thoroughly as hypercubes. Bermond and Peyrat describe however some properties in [BP89]. The most important of these are:

- there exists a simple routing algorithm
- asymptotically optimal loads and forwarding indices
- good broadcasting times
- uniformity and algebraic structure
- fault tolerance
- extendability
- embeddability of other topologies (d-ary tree of height D, shuffle exchange, linear array)
- asymptotically optimal area layout

Binary de Bruijn graphs

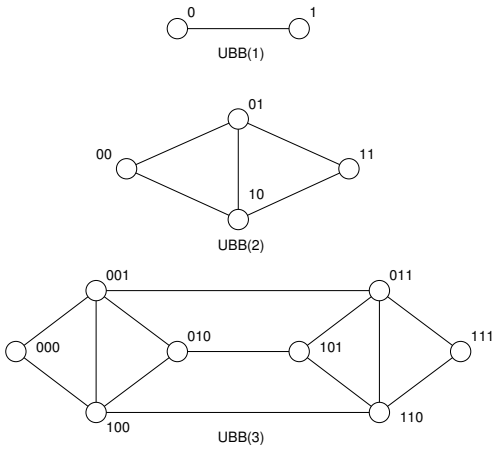


Figure 3.7: Binary de Bruijn graphs

In this project only binary undirected de Bruijn graphs were studied, i.e. graphs of the form $\mathrm{UB}(2, D) \equiv \mathrm{UBB}(D)$. The maximum degree of $\mathrm{UBB}(D)$ is $\Delta = 4$, the diameter is equal to $D = \log_2 |V|$ and the order is $|V| = 2^D$.

3.8 Kautz graphs

Again, after [BP89] a Kautz digraph is defined first.

Definition 3.8.1 *The Kautz digraph* $\mathrm{K}(d, D)$ *is the digraph whose vertices are the words of length* D *over an alphabet of* $d+1$ *symbols, such that any two consecutive symbols are different. There is an arc from* x *to* y *if the* $D-1$ *first symbols of* y *are equal to the* $D-1$ *last symbols of* x*. That is, there is an arc from* $(x_D \cdots x_1)$ *to all vertices* $(x_{D-1} \cdots x_1 \alpha)$*, where* α *is any symbol of the alphabet different from* x_1*.*

□.

$\mathrm{K}(d, D)$ is d-regular, its diameter is D and its order is $d^D + d^{D-1}$.

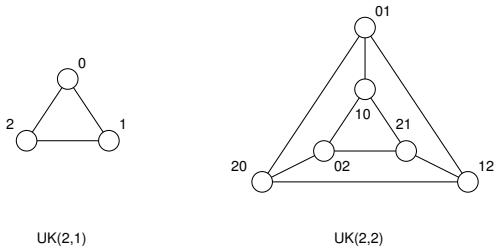


Figure 3.8: Undirected Kautz graphs

Definition 3.8.2 *The undirected Kautz graph* $\mathrm{UK}(d, D)$ *is obtained from* $\mathrm{K}(d, D)$ *by removing the orientations of the arcs and replacing each double edge by a single edge.*

□.

The maximum degree of $\mathrm{UK}(d, D)$ is $2d$, its order is $d^D + d^{D-1}$.

The properties of Kautz graphs are very similar to the properties of de Bruijn graphs given in the previous section.

Chapter 4

Mapping problem

The problem of distributing a parallel program onto a multiprocessor may be viewed in the following way: Suppose several tasks were to execute in parallel, some of which communicate with each other. When assigning the tasks to processors, pairs of tasks that communicate with each other should be placed, if possible, on processors that are directly connected (or are as close as possible one to another).

4.1 Mathematical formulation

Definition 4.1.1 states the mapping problem in a formal way.

Definition 4.1.1 *Mapping problem*
Let the graph of the program (a task graph) to be mapped onto a multiprocessor be denoted $G_T = \langle V_T, E_T \rangle$*, where* V_T *is a set of tasks and each edge* $(u, v) \in E_T$ *denotes that tasks* $u, v \in V_T$ *communicate with each other. Let the processor graph be denoted* $G_P = \langle V_P, E_P \rangle$*, where* V_P *is a set of processors and* E_P *is a set of links connecting processors.*

The mapping to be found is a function: $f_m : V_T \rightarrow V_P$.

The function f_m should minimize the objective function defining the quality of the mapping.

□.

For the choice of different objective functions see [LA87] and [Bok81]. For instance in [Bok81] *cardinality* is used as the objective function.

Definition 4.1.2 *Cardinality $|f_m|$ is defined as the number of task graph edges that fall onto processor graph edges.*

$$|f_m| = \frac{1}{2} \sum_{x,y \in V_T} c_T(x,y) \times c_P(f_m(x), f_m(y))$$

where

$$c_T(x,y) = \begin{cases} 1 & \text{if } (x,y) \in E_T \\ 0 & \text{otherwise} \end{cases}$$

$$c_P(x,y) = \begin{cases} 1 & \text{if } (x,y) \in E_P \\ 0 & \text{otherwise} \end{cases}$$

□.

Many solutions to the mapping problem exist. Some of them differ slightly in the formulation of the problem.

An overview of existing mapping heuristics is presented in this chapter.

4.2 Kernighan-Lin algorithm

This algorithm was designed with a different application in mind, and so its formulation is different than in Definition 4.1.1. It was only later that this algorithm was used to solve the mapping problem.

The Kernighan-Lin (**KL**) algorithm ([KL70]) solves the following problem:

Given a graph G with costs on its edges, partition the nodes of G into subsets no larger than a given maximum size, so as to minimize the total cost of the edges cut.

Kernighan and Lin in [KL70] analyze different approaches and describe why they cannot be used:

Exhaustive Search requires a very large amount of computation and even for some reasonably sized processor graphs is out of the question.

Random Solutions — it is not suitable because typically there are very few optimal partitions, so that the probability of success on any trial can be extremely low.

Max Flow Min Cut — the Ford-Fulkerson algorithm cannot be used to solve this problem because, although it finds quickly a minimal cost cut, there is no way of constraining the sizes of the subsets.

Clustering tries to find clusters, i.e. groups of nodes which are strongly connected in some sense. Again it is not suitable because it is difficult to constrain the sizes of the subsets.

The basic **KL** algorithm finds two subsets of equal size of a given graph of an even number of vertices, trying to minimize the cost of the cut. This problem is certainly simpler than the general one stated before. Given a graph $G = \langle V, E \rangle$, $V = \{v_1, v_2, \ldots, v_{2n-1}, v_{2n}\}$ and a cost matrix $C = (c_{ij})$, the algorithm starts with any initial partition A, B of V and tries to decrease the external cost (i.e. sum of the costs of all edges (v, u) connecting vertices $v \in A$ and $u \in B$) by interchanging subsets X and Y (of the same size) of A and B. If at any step no improvement is possible, the algorithm stops. The subsets $X \subset A$ and $Y \subset B$ are chosen by the following algorithm.

The elements $a \in A$ and $b \in B$ are chosen, such that if they are interchanged, the gain (i.e. the reduction in cost) is maximum. The vertices a and b are stored aside and removed from the subsets A and B. This step is repeated for the new

subsets A and B until they become empty. Then X and Y are chosen to obtain the largest possible gain.

Experiments showed that the results of this algorithm are satisfactory. The time complexity however can be improved. The authors present two ways of speeding up the procedure. Both of them try to find faster the pair a and b of vertices to be exchanged. The faster of the two methods gives the overall running time of $O(n^2)$.

Modifications of the base algorithm

Unequal-Sized Subsets to make the above algorithm work for two subsets of unequal size it is enough to make the starting partition into subsets of required sizes and to restrict the maximum number of pairs to be exchanged in one pass of the procedure.

Elements of Unequal Size each node of size $k > 1$ can be modeled by a cluster of k nodes connected by edges of high cost.

Multiple-Way Partitions a 2-way partitioning procedure is used recursively until the required number of subsets is generated, thus a pairwise optimality is achieved. It does not guarantee, of course, that the final partition is optimal.

Finally ways of generating good starting partitions are discussed.

Fiduccia-Mattheyses algorithm

The modified version of **KL** algorithm was proposed by Fiduccia and Mattheyses in [FM82]. The linear time complexity of this version of the algorithm was achieved by using more efficient data structures and vertex displacement instead of exchanges. The ability to handle graphs with variable vertex weights was a further improvement.

4.3 Parallel version of the Kernighan-Lin algorithm

A parallel version of the algorithm described in Section 4.2 was given by Gilbert and Zmijewski in [GZ87]. The algorithm presented in this article is suitable for distributed memory machines.

The vertices of graph $G_T = \langle V, E \rangle$ are partitioned among $p \geq 2$ processors in a roughly even manner. It is assumed that G_T is a large sparse graph, so that it can be efficiently stored as a collection of adjacency lists. Furthermore all edges have cost one, because for the considered application only number of edges connecting subsets matters. For each vertex assigned to a given processor a list of adjacent vertices is stored in the local memory of the processor.

The algorithm begins by dividing the p processors into two sets P_1 and P_2 of equal size (of course if p is odd, the sizes differ by 1). The partition of vertices induced by P_1 and P_2 is the initial partition. The external cost of the partition is then minimized by a simplified version of the **KL** algorithm. First one processor is selected in each part, say $l_1 \in P_1$ and $l_2 \in P_2$ to be the leader of that part.

Each processor computes the D values ([KL70] of all its vertices and sends these values to its leader. Next each leader selects a vertex with the largest D value. These two vertices are the candidates to be interchanged. The two vertices are marked and stored aside, then the leaders inform each other about their choice. This step (beginning with updating D values) is repeated for the unmarked vertices. When the loop terminates, the leaders decide which vertices to swap.

The procedure is repeated until no further gain can be achieved. Next, the algorithm is recursively applied in parallel to P_1 and P_2.

The authors show that the computational complexity of the algorithm is

$$O(e \log n \log p)$$

where n is the number of vertices and e is the number of edges of G_T.

The total number of messages passed is $O(n \log p)$ and they contain the total of $O(\max(n \log^2 p, e \log p))$ integers.

4.4 Recursive mincut bipartitioning

This is another modification of the **KL** algorithm. Recursive mincut bipartitioning (**ARM**) was proposed by Ercal et al. in [ERS90]. **ARM** is very efficient but the processor graph must be in a form of a hypercube. **KL** and **ARM** use the same optimality criterion, i.e. the total weighted interprocessor communication cost under the mapping, subject to the constraint that the computational loads on the processors be balanced to within a specified tolerance.

The essential idea of the algorithm is to make partial processor assignments to the vertices of the task graph during the recursive bipartitioning steps. At level k in this process, for each vertex, the kth bit of the address of its processor assignment is determined.

The first bipartitioning of the task graph separates the vertices into two groups, each to be assigned to a distinct subcube of the order $\frac{m}{2}$, i.e. the highest-order bit of the processor to which a vertex is to be assigned is uniquely determined. At each succeeding level, during bipartitioning, edge costs are weighted by the number of differing bits in the partial processor assignment of the relevant vertices.

In [ERS90] **ARM** was compared with simulated annealing. The quality of solutions produced by the recursive mincut was within 10% of the solutions from simulated annealing, but required less than one-hundredth the computation time.

The time complexity of **ARM** is $O(n \log_2 m)$, where n is the number of tasks, and m is the number of processors in the hypercube.

However, it must be underlined once more that **ARM** can be used for mapping task graphs onto hypercubes only.

4.5 Pairwise interchange algorithm

In this algorithm, due to Bokhari ([Bok81]), the assumption is made that $|V_T| = |V_P|$ (see Definition 4.1.1), i.e. numbers of tasks and processors are equal (of course if $|V_T| < |V_P|$, a task graph can be adjusted by adding a suitable number of dummy vertices). The algorithm starts with a random assignment of V_T to V_P, and then iteratively interchanges pairs of vertices if the interchange leads to a gain in the cardinality (see Definition 4.1.2) of the mapping. If no interchange leading to a gain is found, a subset of V_T is randomly interchanged.

The time complexity of this algorithm for a class of multiprocessors called *finite element machines* (eight-neighbours mesh) is $O(n^3)$, where n is the number of tasks (and the number of processors).

This heuristics performed well for small-sized problems. The large time complexity of the algorithm makes it, however, not suitable for bigger problems.

More realistic objective functions

A modification of this algorithm is presented in [LA87]. Two improvements were proposed:

1. The initial mapping rather than being random is generated by a heuristic.
2. Four different objective functions were designed. Each of which is appropriate for a different class of problems.

Again the restriction that $|V_T| \leq |V_P|$ applies.

4.6 Graph augmenting heuristics

The use of a graph augmenting approach as a tool for mapping task graphs onto processor graphs is described in [PW87].

Augmenting the interconnection network, i.e. adding some auxiliary edges that decrease distances between the processors, results in a processor graph much more suitable for a given program.

The authors consider the simplest possible interconnection pattern between processors: a one-dimensional linear array. To find out a minimum set of edges needed to augment a line-like multiprocessor, they make use of the concept of an *optimal path cover*. If $G_T = \langle V_T, E_T \rangle$ is a task graph, an optimal path cover S_{G_T} of G_T is a set of vertex disjoint paths that cover all the vertices of G_T and has the maximum possible number of edges.

Two main results are described in the paper:

1. An algorithm to find efficiently an optimal augmentation of the line-like multiprocessor, given an optimal path cover of the program graph.

2. A fast algorithm for finding an optimal path cover of a given graph. This problem is in general NP-complete. The algorithm presented in the article solves the problem for a subclass of graphs called *1-cacti*. A 1-cactus is a simple undirected graph where no vertex lies on more than one cycle.

The above algorithms can be combined into a single algorithm which given a 1-cactus finds an optimal augmentation.

The time complexity of this algorithm is $O(n)$, where n is the number of tasks. Task graphs are restricted, however, to be 1-cacti. This algorithm, rather than mapping a task graph onto an arbitrary processor graph, augments a line-like multiprocessor. The authors don't give any hint as to how useful this approach is in practice.

4.7 Comparison of mapping heuristics

The article [CSG89] evaluates several mapping heuristics. As some of the heuristics work only with hypercube processor graphs, the comparison was restricted to the

mapping onto hypercube multiprocessors.

In the course of comparison experiments some of the algorithms were improved (e.g. by using an efficient data structure called a *bucket list*) and a new fast (linear time) greedy heuristic was proposed.

The following heuristics are described in the article:

- Chen's Greedy heuristics (**G**)
- Simple Greedy heuristics (**SG**)
- All-Swaps Steepest Descent Local Search (**LS**)
- Cube-Heighbours Steepest Descent Local Search (**LSC**)
- Kernigham-Lin Algorithm (**KL**)
- Recursive Mincut Bipartitioning (**ARM**)
- Cube-Neighbour Swaps Simulated Annealing (**SAC**)

In addition to the above basic heuristics, combinations of two algorithms (one for obtaining the initial solution and the other for improving it) were evaluated: **G + LS**, **G + LSC**, **G + KL**, **ARM + LS**, and **SG + LSC**.

The fastest method — **SG** — gives solutions of the worst quality. **SAC** which gives the best results is the slowest heuristics. The exact results in terms of the quality of the graph embedding and the speed of procedures vary depending on the task graphs. Four different kinds of task graphs were used to evaluate the heuristics: random graphs, geometric graphs, trees and hypercubes.

Chapter 5

Scheduling

5.1 Definitions

Independent tasks

Definition 5.1.1 *Scheduling a set of independent tasks*
Given a set of independent tasks with different processing times $T = \{t_1, \ldots, t_n\}$, *and a set of processors* $P = \{p_1, \ldots, p_m\}$, *find an assignment of the tasks to the processors so as to minimize the total execution time of all the tasks.*

□.

This problem is proved to be *NP*-complete. It is highly unlikely then that a polynomial-time algorithm solving the problem exists. However, many polynomial-time heuristics giving good results have been described.

Gonzalez et al. in [GIS77] show that their heuristic gives schedules with a finish time at most twice the optimal finish time.

[HS88] is one of the recent papers on this subject. In this article a polynomial-time, $(1 - \frac{1}{m})$-approximation algorithm, i.e. an algorithm which delivers a schedule

that finishes within at most $(1-\frac{1}{m})$ times the optimal schedule length, is described. Hochbaum and Shmoys showed that for any fixed $\varepsilon > 0$ there exists a polynomial time ε-approximation algorithm. This result is very good, but the problem stated in Definition 5.1.1 is not too practical.

Tasks with precedence constraints

The problem that arises in many applications is the scheduling of tasks constrained by precedence relations. Task x is an immediate predecessor of task y if task x has some data to transfer to task y. For easier analysis it is assumed that data transfer starts after task x terminates, and task y commences execution only after receiving data from all its immediate predecessors. Existing algorithms differ in the level of refinement of this model.

A *task graph* is a convenient way of representing programs for this type of schedulers. A typical definition of task graphs includes sizes of individual tasks and messages exchanged between tasks.

Definition 5.1.2 *A task graph* $G_T = \langle V_T, E_T \rangle$ *is a dag with* n *vertices,* $V_T = \{v_1, v_2, \ldots, v_n\}$.

A function INS : $V_T \rightarrow R$ *defines for each task its size.*

A function M : $E_T \rightarrow R$ *defines for each edge* $(u, v) \in E_T$, *sizes of messages passed between tasks* u *and* v.

□.

If all the processor speeds are the same, a task size INS(v) is equivalent to the computation time of task v.

If all links connecting processors have the same capacity (communication speed) and no initialization times are assumed, the message size M((u, v)) is equivalent to the time required to transfer the message from u to v over a single link.

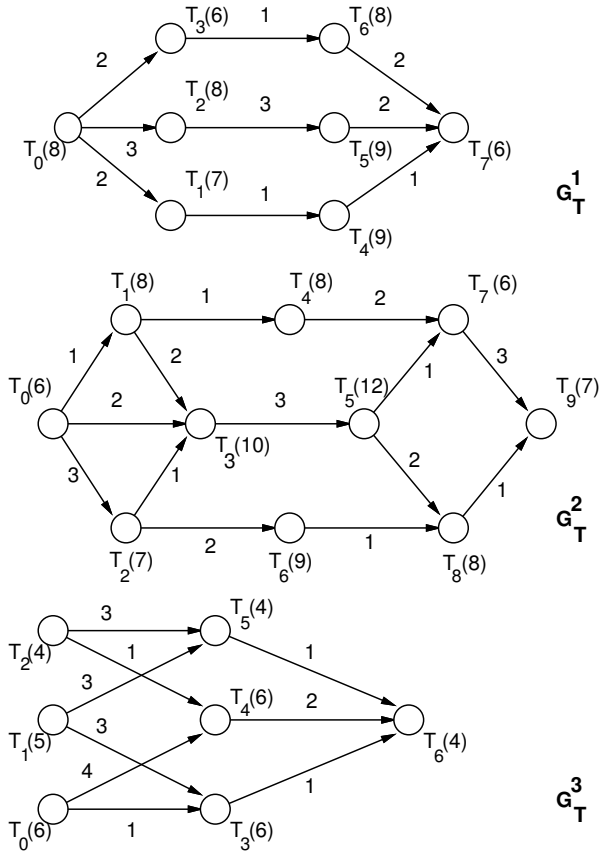


Figure 5.1: Task graphs

The communication time is zero, if tasks u and v are assigned to the same processor. However, if tasks u and v are assigned to different processors, then the communication time depends on the multiprocessor model.

Virtual Distributed System described in [Chr89] is an example of a much simplified model. In this model the communication time is equal to a strictly positive fixed value, if the tasks are assigned to distinct processors, irrespective of the distance. Any two processors may transfer data via a direct communication link and the number of simultaneously transferred messages on link is unlimited. This model is an oversimplified one because the program execution time is independent of the multiprocessor topology.

The same model is used, for example, in [Dar91]. Some of the older papers (e.g. [ACD74]) simplify the model even further, assuming that communication times are always zero, i.e. the function M is constant: $\mathrm{M}(e) = 0, \forall e \in E_T$.

A more sophisticated model of a multiprocessor is used for instance in [ERL90], [MT91], and [HCAL89].

Definition 5.1.3 *A processor graph* $G_P = \langle V_P, E_P \rangle$ *is an undirected graph with* m *vertices* $V_P = \{v_1, v_2, \ldots, v_m\}$ *representing processors.*

An edge (u, v) *exists between vertices* u *and* v *if and only if processors represented by* u *and* v *are connected by a direct link.*

A function $\mathrm{S} : V_P \rightarrow R$ *defines the speed of each processor.*

A function $C : E_P \rightarrow R$ *defines a capacity of each link (communication speed).*

A function $I : E_P \rightarrow R$ *defines the initialization time of each link.*

□.

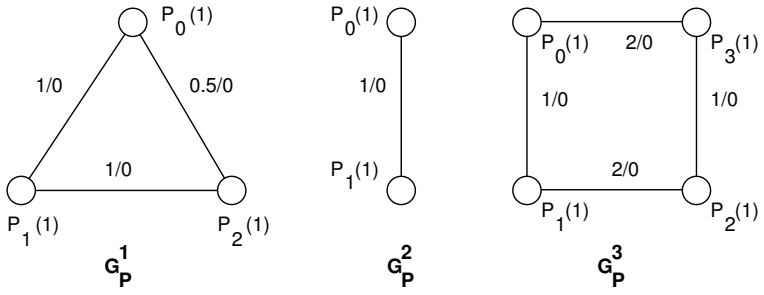


Figure 5.2: Processor graphs

It is assumed that a processor can execute a task and communicate with other processors at the same time.

Small modifications of this model are possible. E.g. in [MT91] link initialization times are assumed to be zero. On the other hand the model of Definition 5.1.3 is extended to take into account memory requirements of the tasks.

Definition 5.1.4

Memory requirements of the tasks are defined by a function $\mathrm{MR} : V_T \rightarrow N$.

Memory capacities of the processors are defined by a function $\mathrm{MC} : V_P \rightarrow N$.

A task $v_T \in V_T$ *may be assigned to a processor* $u_P \in V_P$ *if* $\mathrm{MR}(v_T) \leq \mathrm{MC}(u_P)$, *i.e. memory capacity of the processor* u_P *is at least as big as the memory required by the task* v_T.

□.

A scheduler takes two inputs: a task graph and a processor graph. It produces as an output a pair of values for each task:

- a processor number
- a starting time of the task

Gantt chart is a pictorial representation of this output. For example the task graph G_T^1 (Figure 5.1) scheduled onto a multiprocessor represented by a processor graph G_P^2 (Figure 5.2) could result in a Gantt chart as shown on Figure 5.3.

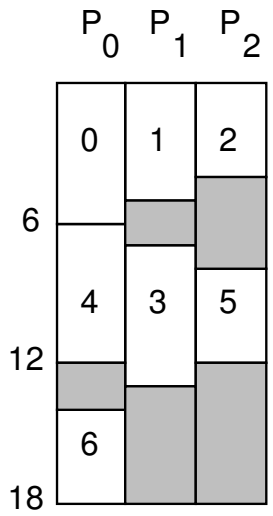


Figure 5.3: A Gantt chart

During the literature survey three papers describing heuristics using the model defined in Definitions 5.1.2 and 5.1.3 were found. Overviews of these heuristics will be given in the rest of this chapter.

5.2 Earliest Task First algorithm

The heuristic called **ETF** (Earliest Task First) was proposed by Hwang et al. in [HCAL89]. This is an event-driven algorithm.

Two assumptions are made:

1. Initialization times (cf. Definition 5.1.3) are assumed to be zero.
2. Links are assumed to be contention free.

The algorithms maintains a set A of available tasks (i.e. tasks with all predecessors already scheduled) and a set I of free processors. For each task, t, its earliest possible starting time $e_s(t)$ is calculated. The starting time is determined by the time when its predecessors are finished, the size of the messages from its preceding tasks and distances between processors assigned to the task and its predecessors. The earliest starting time among all available tasks, $e_s^\wedge$, is calculated as follows:

$$e_s^\wedge = \min_{t \in A}(e_s(t))$$

The task t, for which $e_s(t) = e_s^\wedge$, is scheduled first. If there are more such tasks, the conflict is resolved by using a priority assigned to each task. If the priority list is obtained by a critical path analysis, such a strategy is called **ETF/CP**.

The time complexity of **ETF** is $O(n^2m)$ [1], where n is the number of tasks, and m is the number of processors.

5.3 Depth-first breadth-next heuristic

This algorithm, described by Manoharan and Thanisch in [MT91], uses a technique called depth-first breadth-next (**DFBN**) search to find a task ordering. A program

[1] This doesn't take into account a calculation of distances between processors. A more accurate formula is $O(m(n^2 + m))$ if distances are precalculated, or $O(m(n^2 + m^2))$ if they are computed each time the algorithm is executed.

and a multiprocessor are modeled as in Definitions 5.1.2, 5.1.3, and 5.1.4. The initialization times are assumed to be zero.

DFBN tries to satisfy two properties:

DP1 — assignment of independent tasks to different processors.

DP2 — assignment of dependent tasks to the same processor.

In the algorithm the task graph is searched using the depth-first breadth-next search. Tasks in one path are assigned to the same processor (DP2). Whenever a new path is started, a processor with minimum load is chosen. The load of a processor is simply the sum of execution times of all the tasks that had been assigned to the processor.

Two priority lists: of tasks and of processors are used to resolve conflicts. The priority of a processor is just a distance from the 'most capable' processor (it is chosen by giving consideration to either the number of links it has or the overall link capacity of the processor). Several factors determine task priorities:

1. Tasks with long execution times must get priority.
2. Tasks with large communication requirements must get priority.
3. Tasks with large numbers of successors must get priority.
4. Tasks with long-length successors must get priority.
5. Tasks with large memory space requirements must get priority.

The time complexity of this algorithm is $O(n \log_2 m + e + m^2)$.

5.4 Mapping Heuristic

The Mapping Heuristic (**MH**) was proposed by El-Rewini and Lewis in [ERL90]. Programs and multiprocessors are modeled according to Definitions 5.1.2 and 5.1.3.

This algorithm belongs to a class of list schedulers using an HLFET (Highest Level First with Estimated Times) scheme of assigning priorities ([ACD74]). A priority assigned to each task is just a *critical path* of this task. Because communication is taken into account, the critical path of a task is not static and may change according to the mapping. Critical paths are calculated when the mapping is not known, so it is assumed that all messages are sent through a one-hop channel.

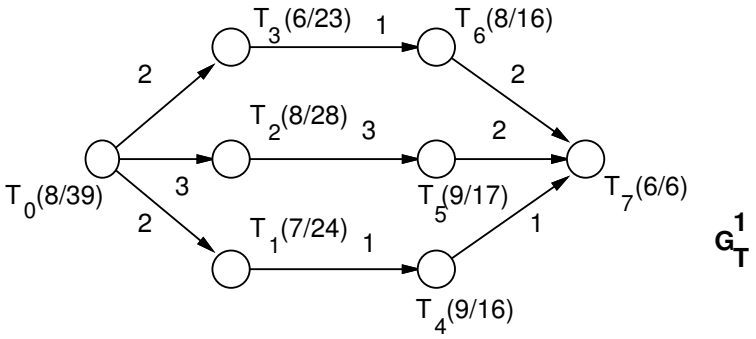


Figure 5.4: Critical paths

The value of the critical path for each task is called its *level*. Figure 5.4 shows the graph G^1_T from Figure 5.1, the first value in parentheses after the name of the task is the task size, the second value is the task level.

MH is an event-driven algorithm. There are four type of events:

(1) task t is ready

(2) task t is done

(3) a message needs to be sent from p_1 to p_2

(4) a message sent from p_1 arrived to p_2

The event list is initialized by inserting the event "task t is ready" at time zero for every task t without predecessors. Events (1) and (2) are sorted according to levels of their tasks. Whenever an event "task t is ready" occurs, the task t is assigned to this processor which ensures the earliest possible completion of t. When "task t is done" occurs, all successors of t are tested for readiness. For each

task t_1 which becomes ready, an event "t_1 is ready" is inserted in the event list. Events (3) and (4) are used for handling contention.

MH is the only heuristic which takes into account contention while generating the schedule.

The time complexity given by El-Rewini and Lewis for their algorithm is $O(n^2m^3 + n^2m + m^3)$ which could be rewritten as $O(em^3 + nm)$ which in turn is equivalent to $O(em^3)$, because $n = O(e)$. As before, n is the number of tasks, e — size of the task graph, and m — number of processors.

Comment on the time complexity of the calculation of critical paths

El-Rewini and Lewis in [ERL90] do not give an analysis of the time complexity of the calculation of critical paths which is a part of the **MH** algorithm. In the case of a task graph, calculating levels of tasks is equivalent to the problem of finding the shortest paths from one of the vertices to all other vertices. In the above calculation of the time complexity of **MH**, an assumption was made that a linear time ($O(n + e)$) algorithm for calculating shortest paths was used ([CLR90]). It is surprising however that some of the papers recommended using the Dijkstra algorithm for calculating longest/shortest paths in a dag. These papers include a classical book on graphs [Deo74] and a very recent article on critical path analysis [KBG90]. The time complexity of the Dijkstra algorithm is $O(n^2)$ or, for sparse graphs, $O(e \log n)$. If this was the algorithm used by El-Rewini and Lewis then, of course, my remark about equivalence of $O(n^2m^3 + n^2m + m^3)$ and $O(em^3 + nm)$ is not valid. This would make **MH** much slower then **DFBN** or **MMH**.

Chapter 6

Modified Mapping Heuristic

MMH — a modified version of the Mapping Heuristic is presented in this chapter. Both heuristics use a concept of a *critical path*. **MH** uses additionally an event list — task which became ready first are scheduled first. In **MMH**, critical paths are the only criteria which decide about the schedule.

6.1 Selecting a scheduler for the project

The aim of this project was to investigate what speedups can be achieved by distributing the same task graph onto multiprocessors with different topologies. So, the first requirement for the scheduling heuristic was that it must take into account an interconnection pattern of the multiprocessor. Of the three heuristics (**ETF**, **DFBN**, **MH**) described in the previous chapter, **DFBN** does not satisfy this requirement — only distances between one "most capable" processor and all the other processors are used and their role is secondary.

This left two heuristics: **MH** and **ETF**. To choose between these two, another factor had to be taken into account. It was decided that the algorithm used in this project had to cope with big task graphs, of the order of at least several thousand tasks. So, the second requirement was a low time complexity, preferably

linear ($O(n+e)$) for a constant number of processors. Unfortunately both **MH** and **ETF** are described to have quadratic time complexities. After some analysis, it was established that the complexity of **MH** is in reality linear. At this point it was decided that **MH** or its modification will be used in the project to evaluate properties of different topologies.

6.2 New heuristic

MH takes contention into account, which means that within the **MH** algorithm a routing algorithm is implemented. The routing algorithm proposed by El-Rewini and Lewis is called *adaptive routing*. The disadvantage of this approach is twofold:

- A fixed routing algorithm is used. Thus porting the scheduler to a machine with a different routing mechanism requires the scheduler to be rewritten.
- Additional cost of maintaining contention information is high.

The time complexity of full **MH** is $O(em^3 + nm)$ as compared with $O(em + nm + m^3)$ for a contention-free version of the Mapping Heuristic (**CFMH**). The complexity of **CFMH** is likely to be the best possible. The element m^3 is necessary for calculating shortest distances between processors (e.g. the Floyd algorithm - cf. [AHU87] and [Deo74]). The elements e and n must be present (each vertex and edge of a task graph must be considered at least once), perhaps not multiplied by m, although scope for improvement here seems to be very small.

It is possible, however, to make this algorithm faster with respect to a constant factor. A modification proposed here is such an improvement. This new version of the Mapping Heuristic is called here the *Modified Mapping Heuristic* (**MMH**) The basic idea of the **MMH** is very similar to this of the **MH**. However, less computation is involved and data structures are simpler. In particular an event list is not used.

6.3 The MMH algorithm

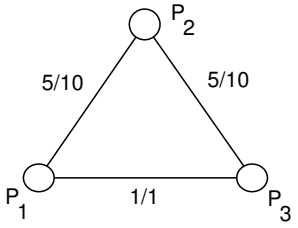
Unlike in the three algorithms described in the previous chapter, the full model of Definitions 5.1.2, 5.1.3, and 5.1.4 is implemented in **MMH**.

Calculating shortest paths

The process of calculating shortest paths between all pairs of processors is not straightforward in this multiprocessor model. Cost of sending a message of size m over a link connecting processors p_1 and p_2 is

$$\frac{m}{C((p_1, p_2))} + I((p_1, p_2))$$

It easy to demonstrate that shortest paths are determined not only by a multiprocessor topology but by a message size as well.



	m=10	m=100
path: (P_1,P_3)	11	101
path: (P_1,P_2,P_3)	24	60

Figure 6.1: Dependency of shortest paths on a message size

In the multiprocessor shown on Figure 6.1, the shortest path between processors p_1 and p_3 is $(p_1 p_3)$ for a message of size 10, and $(p_1 p_2 p_3)$ for a message of size 100. Here, as on Figure 5.2 the first number associated with each link is its speed, the second one is the initialization time.

A typical task graph contains messages of different sizes. This would require maintaining several shortest paths tables for several message size ranges. Moreover computing these message size ranges and appropriate tables seems to be a very complex problem.

There are several ways of simplifying this problem. One, adopted by Hwang et al. in [HCAL89] and by Manoharan and Thanisch in [MT91], is to ignore ini-

tialization times. This approach is very inaccurate if messages are small. The second method, adopted by El-Rewini and Lewis in [ERL90], is to assume that all links are identical and simply uses number of hops to find shortest paths. This approach ignores a very interesting case of distributing programs onto heterogeneous multiprocessors, e.g. networks consisting of separate computers.

The (heuristic) algorithm used in **MMH** first calculates the average message size, $\bar{m}$. The value of $\bar{m}$ is then used to find shortest paths using the Floyd algorithm. While finding shortest paths two tables are generated, SPC containing a sum of inversions of link capacities (i.e. unit transmission times), and SPI containing a sum of initialization times.

If a shortest path between processors p_{i_1} and p_{i_k} is: $(p_{i_1} p_{i_2} \ldots p_{i_k})$ then:

$$\mathrm{SPC}(p_{i_1}, p_{i_k}) = \sum_{j=1}^{n-1} \frac{1}{C((p_{i_j}, p_{i_{j+1}}))}$$

$$\mathrm{SPI}(p_{i_1}, p_{i_k}) = \sum_{j=1}^{n-1} I((p_{i_j}, p_{i_{j+1}}))$$

Both SPC and SPI are assumed to be zero for messages transferred between tasks on the same processor.

$$\mathrm{SPC}(p, p) = 0$$

$$\mathrm{SPI}(p, p) = 0$$

Time of sending a message of size m between processors p_1 and p_2 can be calculated as:

$$t(p_1, p_2, m) = m \times \mathrm{SPC}(p_1, p_2) + \mathrm{SPI}(p_1, p_2)$$

Algorithm structure

The construction of **MMH** is very similar to MH, although, results produced by two heuristics may differ slightly. **MMH** is explained by a following fragment of pseudocode:

```
procedure MMH is
begin
  Load the task graph
  Load the processor graph
  Calculate the level of each task
  Calculate shortest distances between processors
  Initialize list of ready tasks, RTL
  while RTL is not empty loop
    Get task t from RTL
    ScheduleTask(t);
  end loop;
end MMH;
```

A list of ready tasks is maintained throughout the scheduling. It is initialized with all the tasks with no predecessors. The instruction "Get task t from RTL" removes the task with the highest level from the list of ready tasks. The procedure ScheduleTask is as follows:

```
procedure ScheduleTask(t) is
begin
  Proc := p where FinishTime(t,p) ≤ FinishTime(t,i), 0 ≤ i ≤ m
  assign t to Proc
  HandleSuccessors(t);
end ScheduleTask;
```

Function FinishTime(t,p) returns the finishing time of a task t if it is placed on a processor p. This takes into account the finishing time of the last task assigned to p and the time of arrival of all messages from predecessors of t assuming contention-free links.

Procedure HandleSuccessors(t) checks whether any of successors of t becomes ready, i.e. all its predecessors except t are already scheduled. If yes, this task is added to the list RTL.

G_T	G_P	t_{ETF}	t_{DFBN}	t_{MH}	t_{MMH}
G_T^1	G_P^1	37	35	35	35
G_T^1	G_P^2	45	51	42	42
G_T^1	G_P^3	37	35	32	32
G_T^2	G_P^1	59	57	54	54
G_T^2	G_P^2	59	58	54	54
G_T^2	G_P^3	59	57	53	53
G_T^3	G_P^1	18	18	19	18
G_T^3	G_P^2	24	24	22	23
G_T^3	G_P^3	19	19	19	18

Table 6.1: Comparison of scheduling heuristics

The time complexity of **MMH** is $O((n+e)m + m^3)$. The quality of schedules is good compared to other heuristics. Table 6.1 gives execution times of programs scheduled by each of the heuristics: **ETF**, **DFBN**, **MH**, and **MMH**. Task graphs presented on Figure 5.1, and processor graphs presented on Figure 5.2 were used in these tests. The idea of the table and the values for **ETF** and **DFBN** are taken from [MT91].

In all cases schedules generated by **MMH** were at least as good as those generated by **ETF** and **DFBN** heuristics (in 7 out of 9 cases they were better). In almost all cases (8 out of 9) execution times of schedules generated by **MMH** were not longer than those of **MH** (two of the times are indeed shorter).

Chapter 7

Comparing topologies

Experiments comparing speedups achieved for different multiprocessor topologies were a significant part of this project. Several topologies were investigated, some of these were widely known and were described in literature (cf. Chapter 3), others were designed specifically for this project.

In the first two sections of this chapter, task graphs and processor graphs used in the experiments are described. In the rest of the chapter, results are presented and discussed.

7.1 Task graphs

To clearly show differences between topologies, processor graphs must contain at least tens of processors. It was decided to use processor graphs of orders up to approximately one hundred. Task graphs used to measure qualities of such processor graphs had to be of orders of at least a few hundred.

Computationally intensive programs offer different possible speedups than communication intensive programs. Therefore it was necessary to investigate task graphs with various ratios of communication to computation.

These two facts were the reasons for the decision to use random task graphs generated by a program with capabilities to change the above parameters. In the case of task graphs obtained from real programs, arbitrary changes of these parameters would not be possible.

To distinguish between computationally intensive and communication intensive programs, an additional parameter, called the *granularity* of the task graph, was defined.

Definition 7.1.1 *Granularity of a task graph is a ratio of the sum of sizes of all messages to the sum of sizes of all tasks.*

$$G_{\mathrm{T}} = \langle V_P, E_P \rangle$$

$$g(G_{\mathrm{T}}) = \frac{\sum_{e \in E_T} \mathrm{M}(e)}{\sum_{v \in V_T} \mathrm{INS}(v)}$$

□.

The definition has the following interpretation: granularity is a ratio of the time required to send all the messages through a single link to the time required to execute all tasks on a single processor.

The following task graph parameters were fed as input data to the program which generated graphs:

- Minimum (n_{min}) and maximum (n_{max}) numbers of tasks. The actual order of the task graph was picked at random (using uniform distribution) from a range between n_{min} and n_{max}. In all experiments these values were set to $n_{min} = n_{max} = n$.
- Minimum (INS_{min}) and maximum (INS_{max}) task sizes.
- Minimum (M_{min}) and maximum (M_{max}) message sizes. M_{min}, M_{max}, INS_{min}, INS_{max} could be used to obtain the required granularity.

- Maximum number of sources.

Task graphs generated by this program had vertices with out-degree not greater than 10. The interpretation of this condition is that none of the tasks sends messages to more than 10 successors. It seems that task graphs obtained from real programs would be characterized by a constraint of this kind.

7.2 Processor graphs

The following processor graphs were used in the experiments:

1. Complete binary tree: CBT(1), CBT(2), ..., CBT(7).
2. Binary de Bruijn graphs: UBB(1), ..., UBB(7).
3. Kautz 2 graphs: UK(2, 2), ..., UK(2, 6).
4. Kautz 3 graphs: UK(3, 2), ..., UK(3, 4).
5. Hypercube: HC(1), ..., HC(7).
6. Enhanced hypercube: EHC(1), ..., EHC(7).
7. Symmetrical 2-D mesh: MS(2), ..., MS(10).
8. Symmetrical torus: TS(1), ..., TS(10).
9. Ring: R(2), ..., R(6), R(10), R(20), R(30), R(40), R(80)

The topologies CBT(n), UBB(n), UK(2, n), UK(3, n), HC(n), EHC(n), and R(n) have been already described in Chapter 3. MS(n) and TS(n) are the topologies described in Sections 3.2 and 3.3 respectively, with the additional constraint that the width and height are equal: MS(n) $\equiv$ M(n, n), TS(n) $\equiv$ T(n, n).

Torus 2 (T(2, n)) and torus 4 (T(4, n)) have fixed widths of 2 and 4, respectively.

Enhanced complete binary trees

Even before starting the experiments, it seemed obvious that binary trees were not well suited for applications with random traffic. The links near a root of a tree would soon become bottlenecks, as all the traffic between processors in a left subtree and processors in a right subtree would be directed through a root node. Enhanced binary trees were an attempt to prevent this undesirable effect. The obvious way of creating shorter paths between leaves is to connect each level of a tree using one of the topologies described above. The simplest of them — a ring — is used in graphs ECBT5(n).

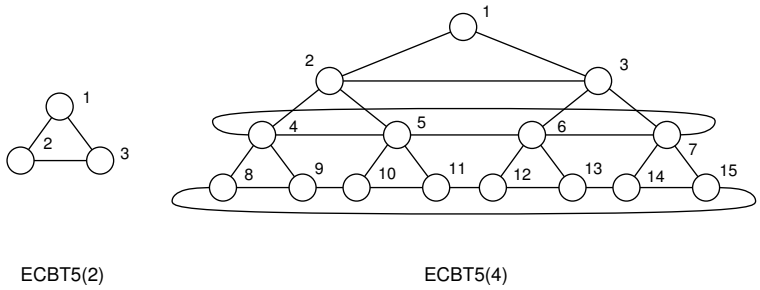


Figure 7.1: Enhanced binary trees ECBT5(n)

This topology is presented in Figure 7.1. For $n \geq 6$ the diameter of ECBT5(n) is equal to the diameter of CBT(n), but the average distance between vertices is smaller, and this value seems to be more important than a diameter. The maximum degree of ECBT5(n) is 5 (hence the name).

Because many other topologies examined in the project have a maximum degree $\Delta = 4$, another version of an enhanced binary tree with a maximum degree equal to 4 was designed, to make fair comparisons possible. In these graphs, called ECBT4(n), 'neighbouring' vertices without common father are connected by an edge. Examples of such trees are shown in Figure 7.2. For $n \geq 5$, the diameter of ECBT4(n) is equal to the diameter of CBT(n), but again the average distance is shorter.

These additional processor graphs were used in experiments:

1. Enhanced complete binary tree 4: ECBT4(1), . . . , ECBT4(7).

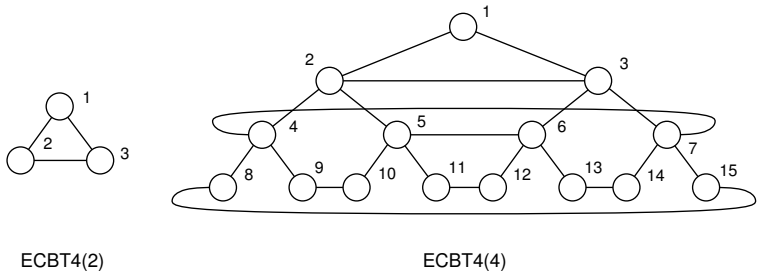


Figure 7.2: Enhanced binary trees ECBT4(n)

2. Enhanced complete binary tree 5: ECBT5(1), . . . , ECBT5(7).

Comparison of processor graphs

13 topologies were used throughout the experiments. Their properties are summerized in Table 7.1. The special role of processor graphs with fixed degrees should be underlined here. In hardware terms, this means a constant number of links per processor. It is a very desirable feature of an interconnection topology. The cost of building a multiprocessor of such components can be considerably lower. Specifically, topologies with a degree 4 or smaller can be implemented directly using transputers, as these have four links.

Topologies used here can be divided into three groups according to the order of their diameters.

- *logarithmic* group
- *square root* group
- *linear* group

It is natural to expect that processor graphs from a group with a lower diameter order should outperform processor graphs with higher diameter orders (at least for greater number of processors).

Topology	Max. degree	Diameter
CBT(n)	3	O($\log_2 \|V\|$)
ECBT4(n)	4	O($\log_2 \|V\|$)
ECBT5(n)	5	O($\log_2 \|V\|$)
UBB(n)	4	O($\log_2 \|V\|$)
UK($2, n$)	4	O($\log_2 \|V\|$)
UK($3, n$)	6	O($\log_2 \|V\|$)
HC(n)	$\log_2 \|V\|$	O($\log_2 \|V\|$)
EHC(n)	$\log_2 \|V\| + 1$	O($\log_2 \|V\|$)
MS(n)	4	O($\sqrt{\|V\|}$)
TS(n)	4	O($\sqrt{\|V\|}$)
T($2, n$)	4	O($\|V\|$)
T($4, n$)	4	O($\|V\|$)
R(n)	2	O($\|V\|$)

Table 7.1: Comparison of topological properties of processor graphs

7.3 Experiments

Experiments were grouped in series. In one series three task graphs with the same granularity were scheduled onto each of the processor graphs listed in the beginning of Section 7.2. The execution of programs scheduled in this way was simulated. In this simulation contention was taken into account. The routing algorithm directed messages through a shortest path. If more parallel shortest paths existed, one of them was picked at random, to distribute the traffic evenly.

The simulator took three inputs: a task graph, a processor graph, and a sched-

ule and produced one value as an output: the execution time t_{par}. The parallel execution time was converted to a speedup as this made comparisons between results for task graphs of different granularities easier.

Definition 7.3.1 *A sequential execution time of a task graph* t_{seq} *is the sum of sizes of all tasks in the graph.*

$$G_{\mathbf{T}} = \langle V_T, E_T \rangle$$

$$t_{\mathrm{seq}} = \sum_{v \in V_T} \mathrm{INS}(v)$$

□.

Definition 7.3.2 *A speedup* σ *of a program is a ratio of the sequential execution time to the parallel execution time of this program.*

$$\sigma = \frac{t_{\mathrm{seq}}}{t_{\mathrm{par}}}$$

□.

Another way of interpreting the result of the simulation is to convert it to a utilization of a multiprocessor, which indicates what percentage of the CPU time was spent on program execution.

Definition 7.3.3 *A utilization* ρ *of a multiprocessor represented by a processor graph* $G_{\mathbf{P}}$ *during the execution of a program represented by the task graph* $G_{\mathbf{T}}$ *is a ratio of the sum of all task sizes to the total reserved time (including idle time) of all processors.*

$$G_{\mathbf{T}} = \langle V_T, E_T \rangle$$

$$G_{\mathbf{P}} = \langle V_P, E_P \rangle$$

$$\rho = \frac{\sum_{v \in V_T} \mathrm{INS}(v)}{|V_P| \times t_{\mathrm{par}}}$$

□.

By comparing Definitions 7.3.1, 7.3.2, and 7.3.3, it is easy to observe that the last formula may be simplified to:

$$\rho = \frac{\sigma}{m}$$

where m is the number of processors.

Utilization is a very important measure of the quality of a schedule as it indicates the time spent in a multiprocessor on doing useful work. It is possible, therefore, to set a lower limit on utilization of employed resources. The exact value is a tradeoff between the speedup and economical considerations.

In the rest of this section results of the experiments will be presented and discussed. The graphs were plotted using the average of results for three task graphs of the same granularity. In this section (as throughout the entire report) n is the number of tasks, e — a number of edges in a task graph, and m — number of processors.

Total of 80 processor graphs were used. They were all assumed to be homogeneous, i.e. all processors were capable of executing any task with the same speed, all links had the same speeds and initialization times.

$$\forall G_{\mathbf{P}} = \langle V_T, E_T \rangle \cdot \forall v \in V_T \cdot \mathrm{S}(v) = 1$$

$$\forall G_{\mathbf{P}} = \langle V_T, E_T \rangle \cdot \forall e \in E_T \cdot C(e) = 1 \wedge I(e) = 0$$

Each series consisted of 240 experiments (3 task graphs scheduled onto 80 processor graphs).

First series: $g = 0.01$, $n = 1000$

In this series of experiments, task graphs of granularity 0.01 were used. Such graphs can be said to represent *computationally intensive* programs, i.e. programs

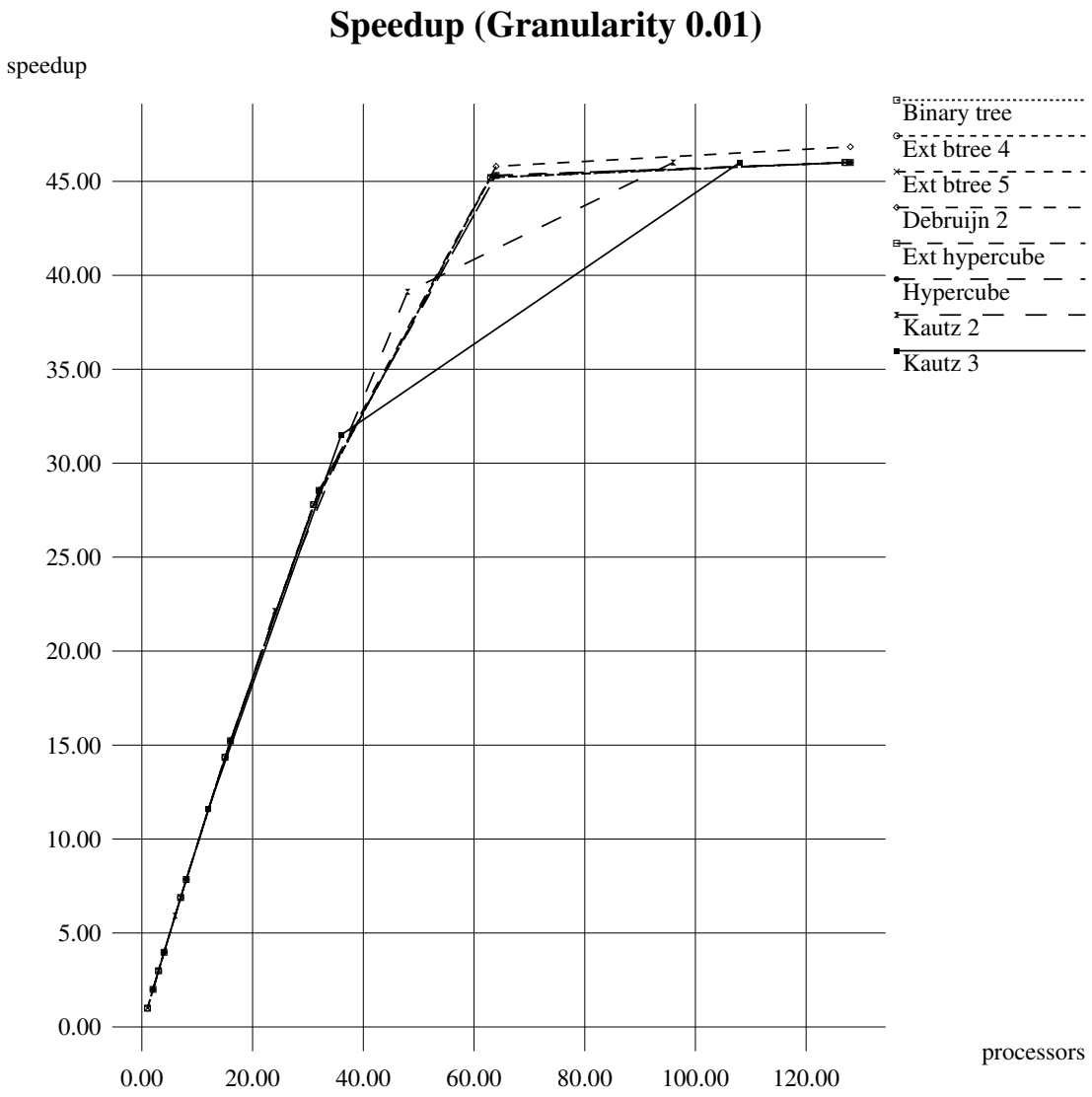


Figure 7.3: Speedups: $g = 0.01$, $n = 1000$

with messages of very small sizes as compared to task sizes. It could be expected that speedups achieved for this granularity would be (almost) independent on the interconnection pattern of a multiprocessor. Indeed, results obtained for all the topologies are almost identical — see Figures 7.3, 7.4, 7.5, and 7.6.

Note that, because 13 topologies were examined in the experiments, it wasn't practical to present results for all topologies in a single graph. Therefore topologies were divided, somewhat arbitrarily, into two groups. The first of them contained all processor graphs with logarithmic diameters, the second — all other processor graphs.

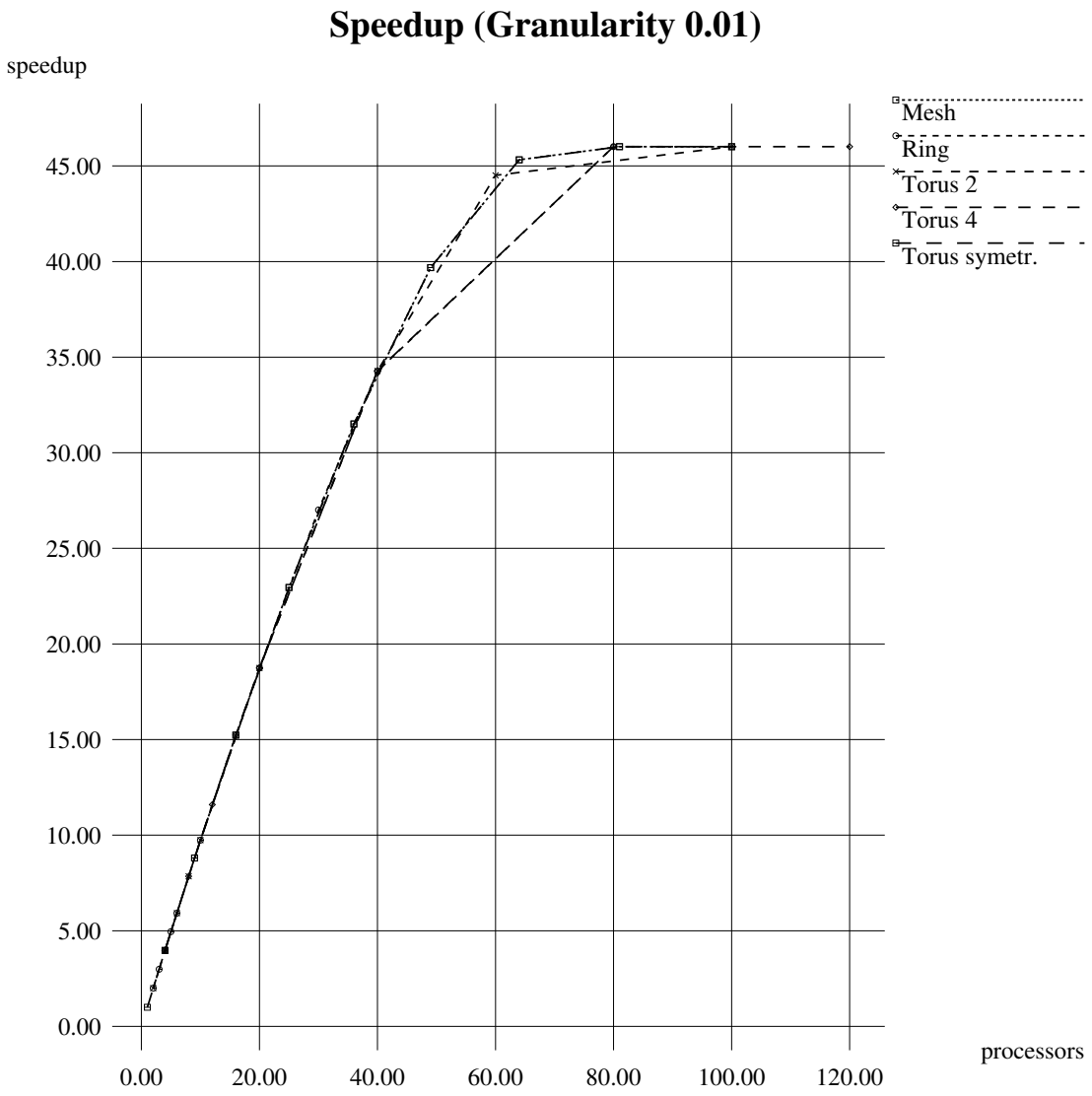


Figure 7.4: Speedups: $g = 0.01$, $n = 1000$

The granularity 0.01 is in this case optimal in the sense that the results for a small granularity present upper bounds for speedups possible to achieve with this scheduler for task graphs of this size and granularity generated in the same way. It could be expected that the results for task graphs of higher granularities would be worse.

Good, quasi-linear increase of speedup was achieved for processor graphs with up to 60 processors. Above 60 processors, the graphs in Figures 7.3, 7.4 saturates and further increases of number of processors only slightly increase speedups.

Furthermore, if we define a 'good result' as a schedule which achieves a uti-

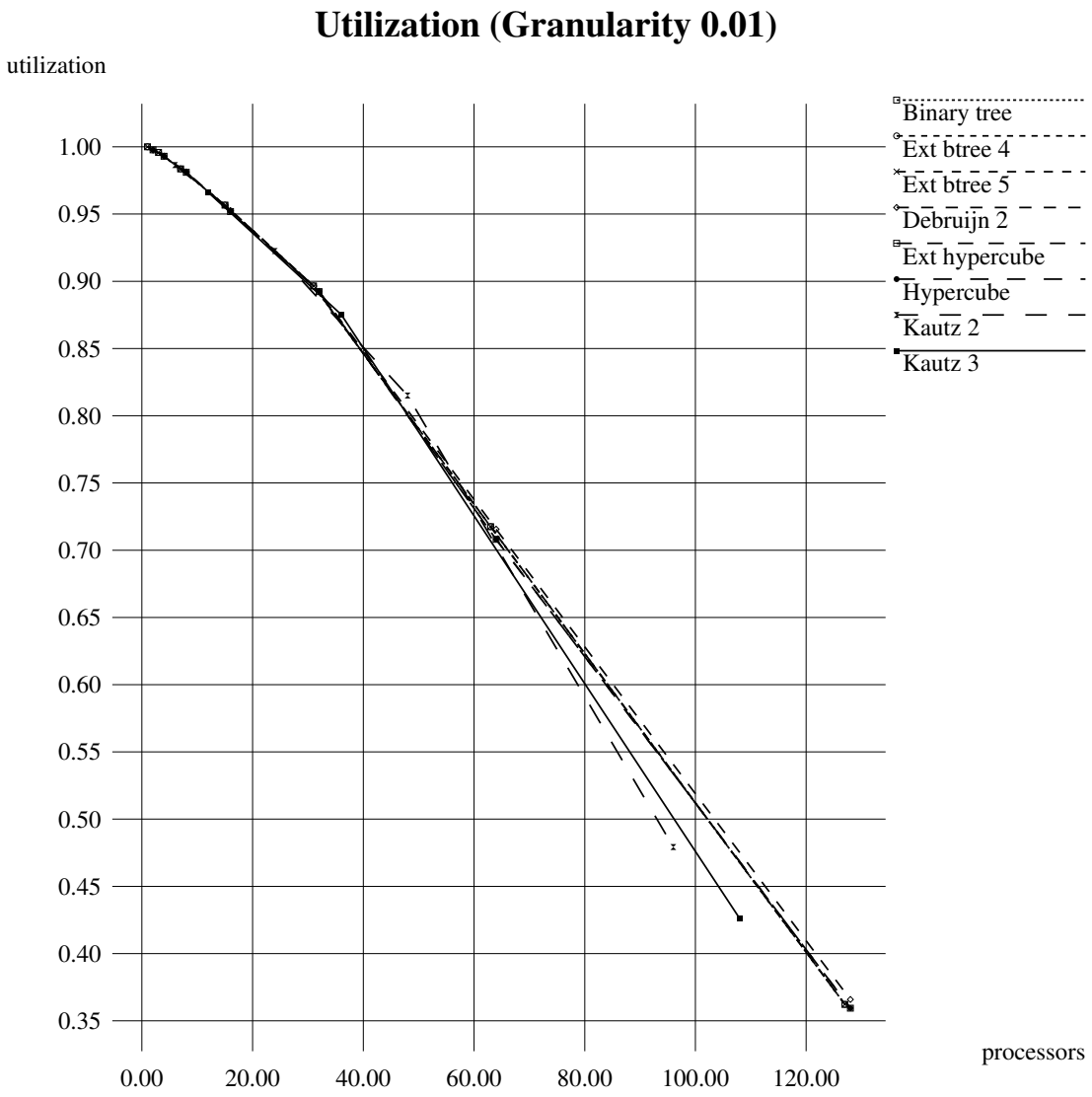


Figure 7.5: Utilization: $g = 0.01$, $n = 1000$

lization of at least 50%, then for all processor graphs, 'good results' were obtained for up 100 processors.

Second series: $g = 0.5$, $n = 1000$

As the granularity is increased for the same number of processors, differences between topologies become visible. Looking at Figures 7.7 and 7.8, we can distinguish two groups of topologies:

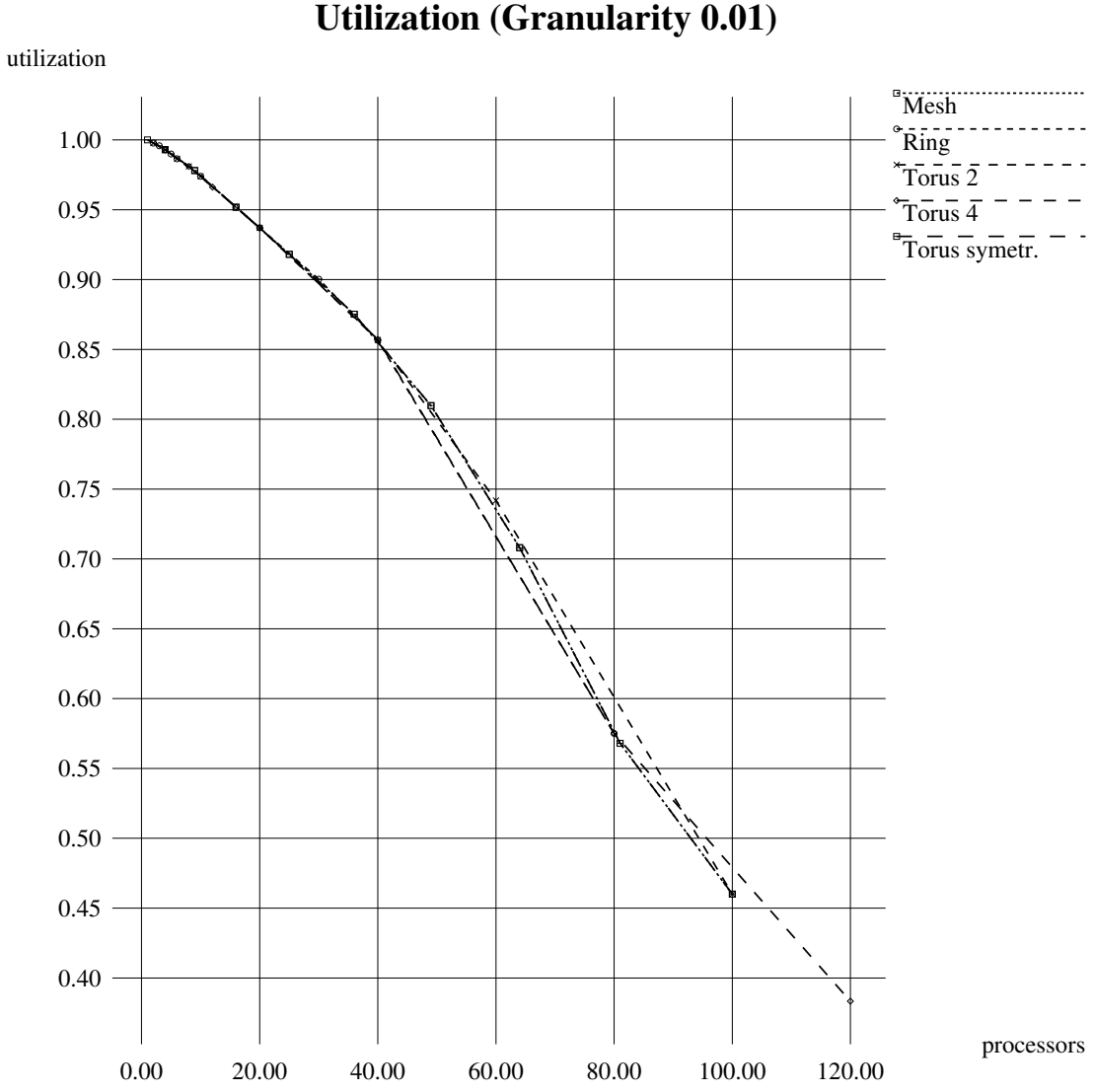


Figure 7.6: Utilization: $g = 0.01$, $n = 1000$

- HC(n), EHC(n), UK($2, n$), UK($3, n$), UBB(n), and TS(n). Speedups for these processor graphs grow relatively fast and almost linearly as the number of processors increases up to 60. Between 60 and 120, speedups grow slowly and reach a maximum of 40–45.

- MS(n), R(n), T($2, n$), T($4, n$), ECBT5(n), CBT(n). Speedups for these topologies occupy a wider spectrum, with maximum speedup ranging from 32 (MS(10)) to 9 (CBT(7)).

Third series: $g = 2$, $n = 1000$

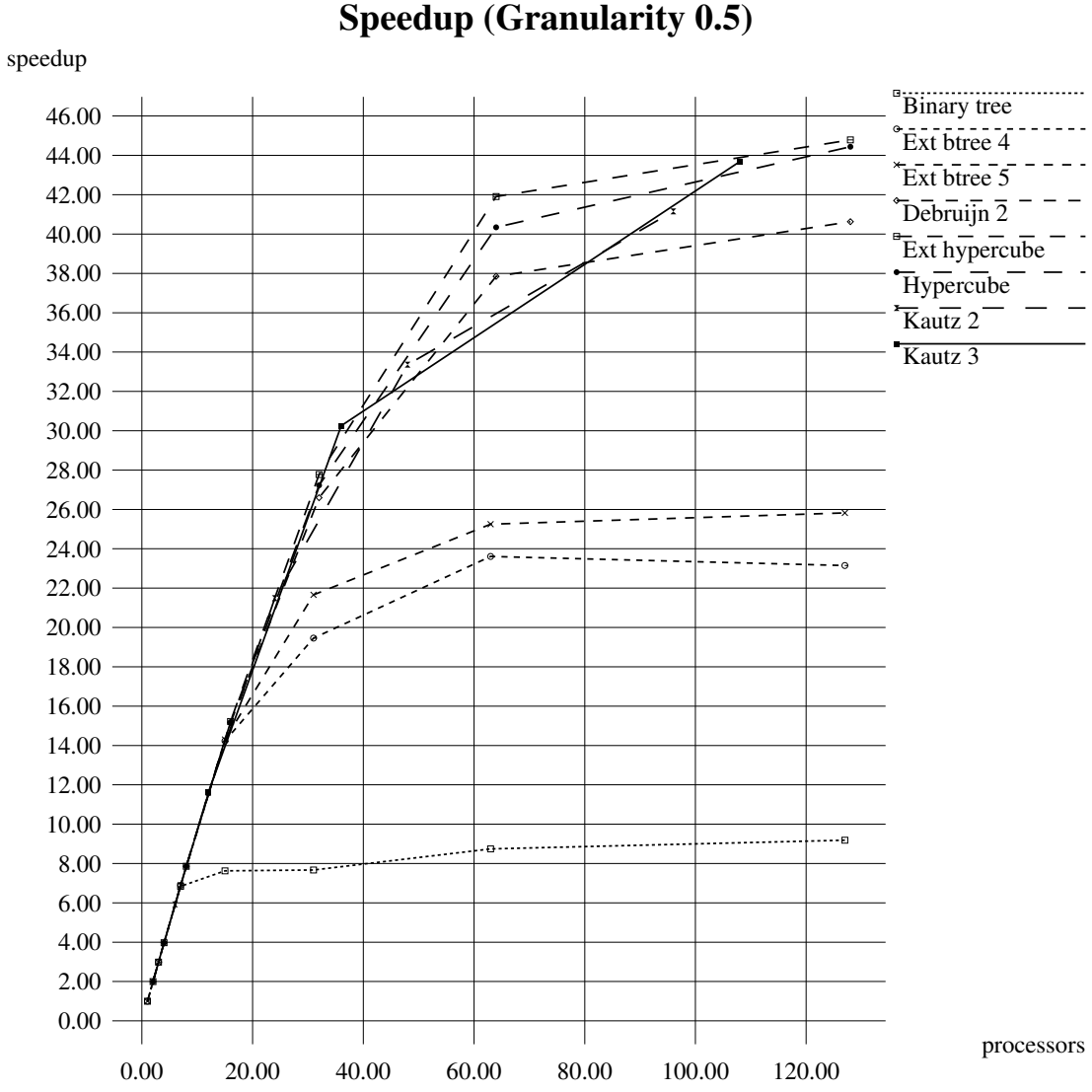


Figure 7.7: Speedups: $g = 0.5$, $n = 1000$

Programs represented by task graphs with granularity 2 may be viewed as programs with time necessary to send all messages over a single link twice as large as the sequential execution time. One can argue whether programs like these should be parallelized in the first place (although in the experiments considerable speedups were achieved). Such extreme cases show, however, more clearly differences between topologies. Figures 7.9, 7.10, 7.11, and 7.12 show results achieved for task graphs with this granularity.

The results for processor graphs with fewer than 30 processors do not show enough differences between topologies to order them according to their "quality" (quality of a processor graph is measured by a speedup achieved for a given task

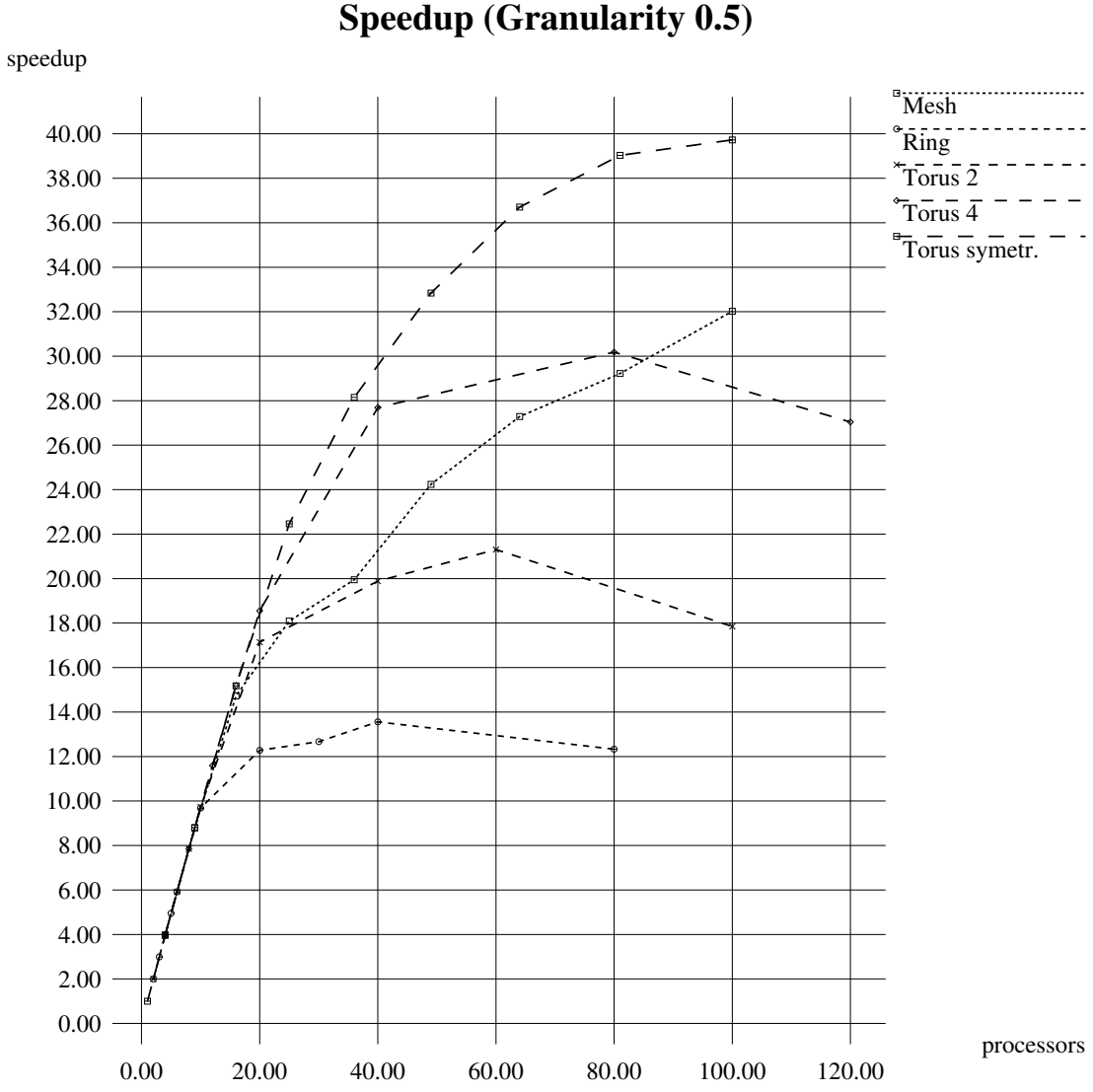


Figure 7.8: Speedups: $g = 0.5$, $n = 1000$

graph). For 30 and more processors, differences become very clear. Accurate comparison is not possible, because most of examined processor graphs are defined only for some numbers of processors. By examining Figures 7.9 and 7.10 it is, however, easy to observe that it can be assumed that a continuous graph of a speedup of a given topology created by connecting with lines subsequent points of is a good approximation of an ideal continuous graph.

Speedups for topologies of the logarithmic group do not decrease as the number of processors grows. On the other hand, the performance of processor graphs of the linear group falls after reaching a maximum. A symmetrical mesh behaves like topologies from the linear group, and a symmetrical torus, like topologies

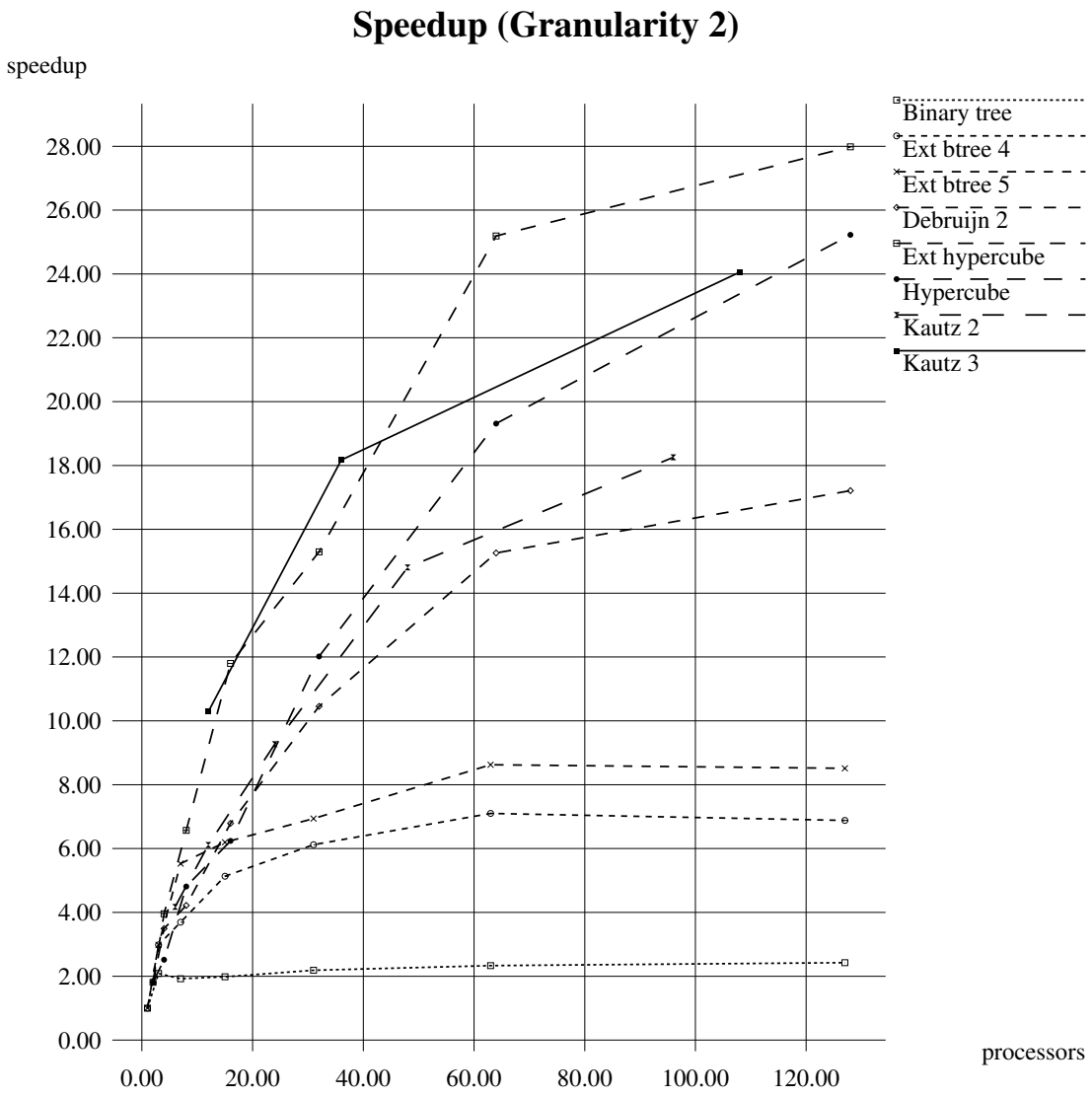


Figure 7.9: Speedups: $g = 2$, $n = 1000$

from the logarithmic group. This is easy to explain for (enhanced) hypercubes and (enhanced) trees, because in these topologies it is possible to find subgraphs with smaller numbers of processors. So if, for example, an optimal schedule for a given task graph scheduled onto a hypercube is obtained for HC(3), the scheduler when generating a schedule for HC(6) can obtain the optimal speedup by using only 8 processors forming a subcube HC(3). Such a subcube is guaranteed to exist (cf. Section 3.5).

This strategy is, of course, not possible for ring-like topologies.

This easy explanation of the phenomenon is not sufficient for a symmetrical

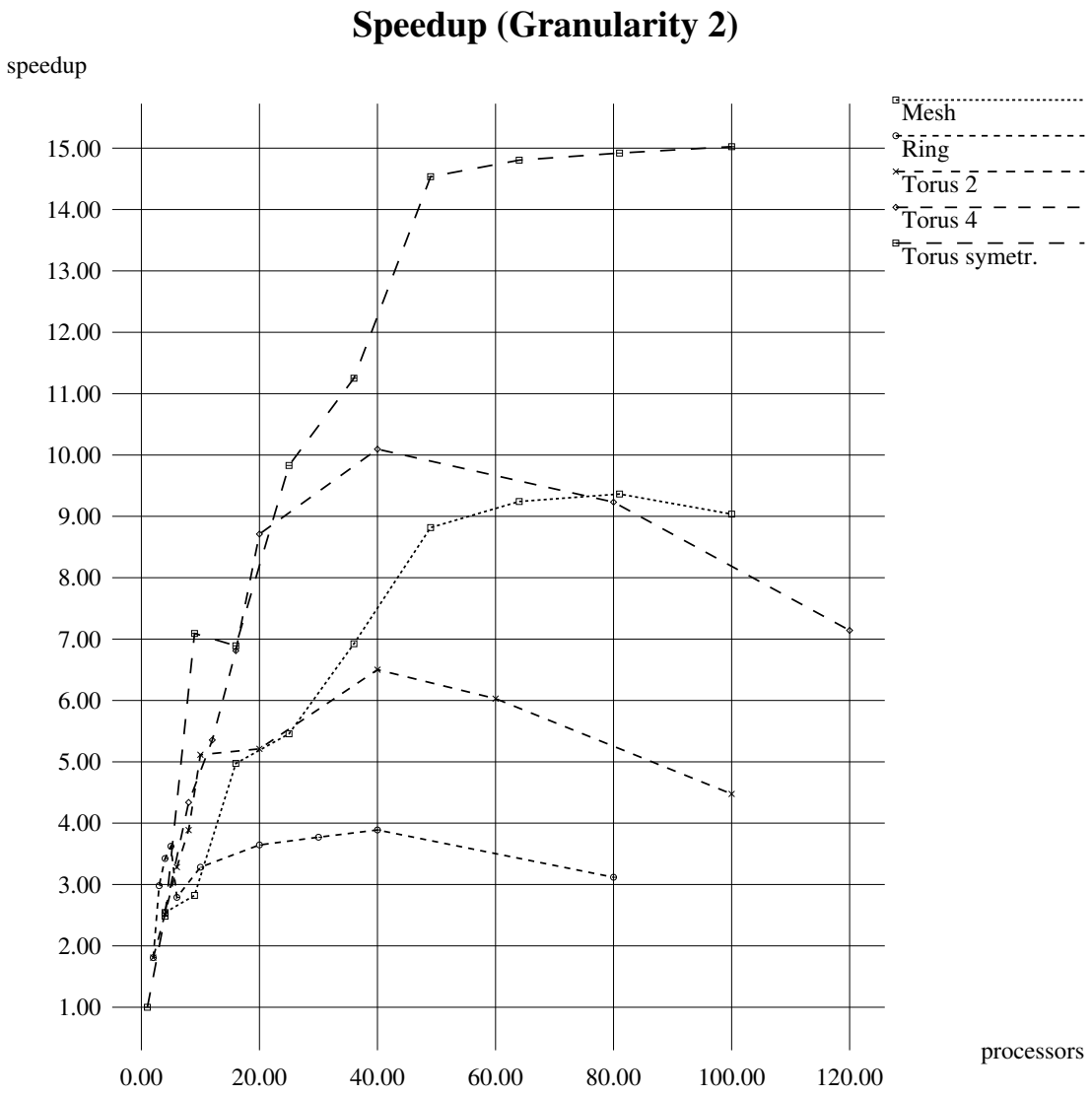


Figure 7.10: Speedups: $g = 2$, $n = 1000$

torus, but, perhaps, a decrease of speedups would be observed for higher numbers of processors.

For 64 processors, the order of the topologies, from the best to the worst, is: EHC(n), UK($3, n$), HC(n), UK($2, n$), UBB(n), TS(n), T($4, n$), ECBT5(n), MS(n), ECBT4(n), T($2, n$), R(n), and CBT(n).

The order is slightly different for different numbers of processors, namely:

- for $m < 40$, UK($3, n$) is better than EHC(n).
- for $m > 80$, MS(n) is better than T($2, n$).

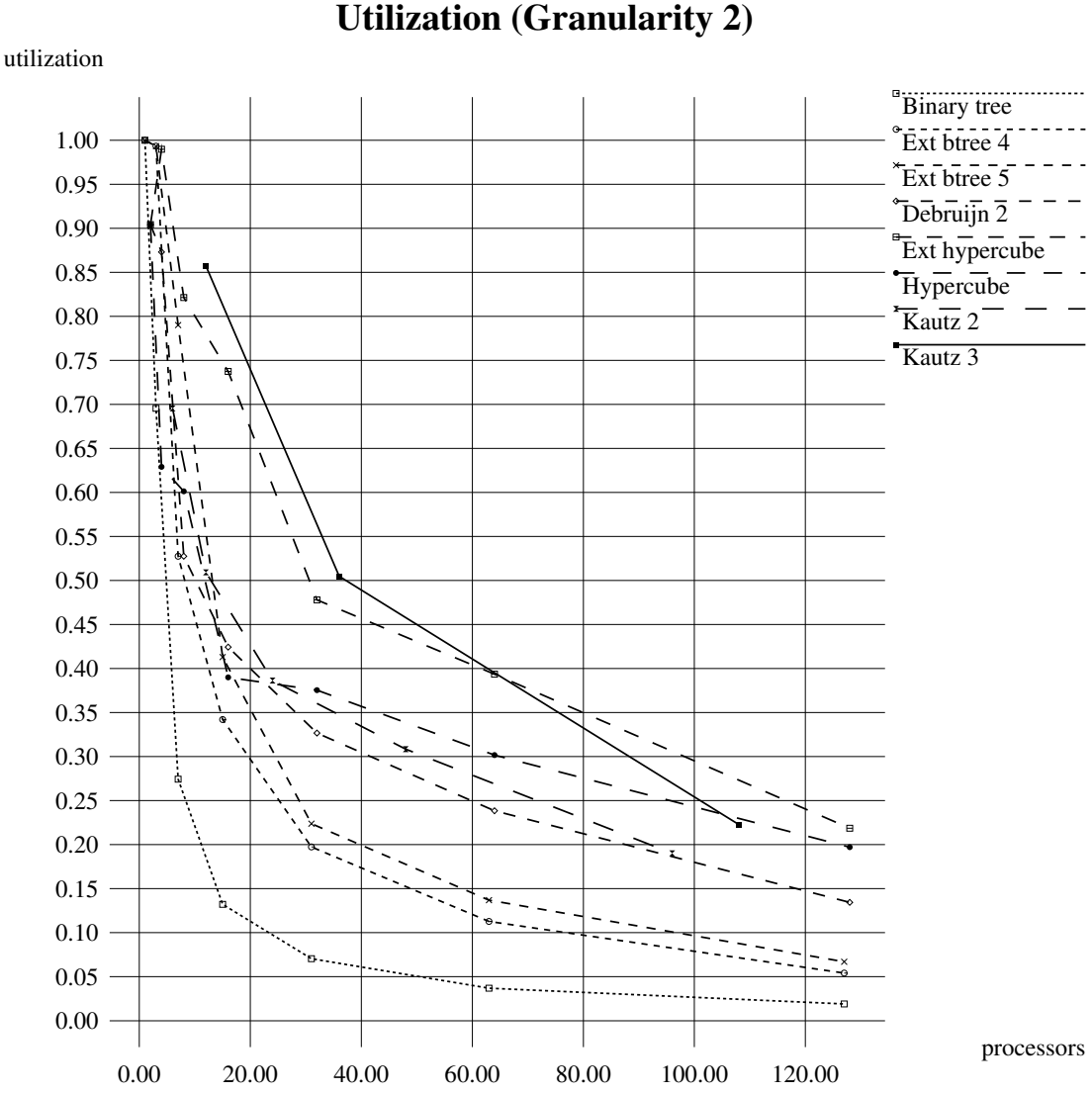


Figure 7.11: Utilization: $g = 2$, $n = 1000$

Apart from these, the above order is valid for all values of m greater than 30.

The results show that binary trees (also the enhanced versions, even though they offer 3 to 4 times better performance), mesh, and ring-like graphs (i.e. R(n), T$(2, n)$, and T$(4, n)$) are not suitable for these types of applications.

The best speedups are obtained for enhanced hypercubes, which is hardly surprising because enhanced hypercubes are the most expensive topologies in the terms of number of links per processor.

Of the topologies with the maximum degree 4, three are evidently best: UK$(2, n)$, UBB(n), and TS(n). Differences between them are only minimal, so the choice is

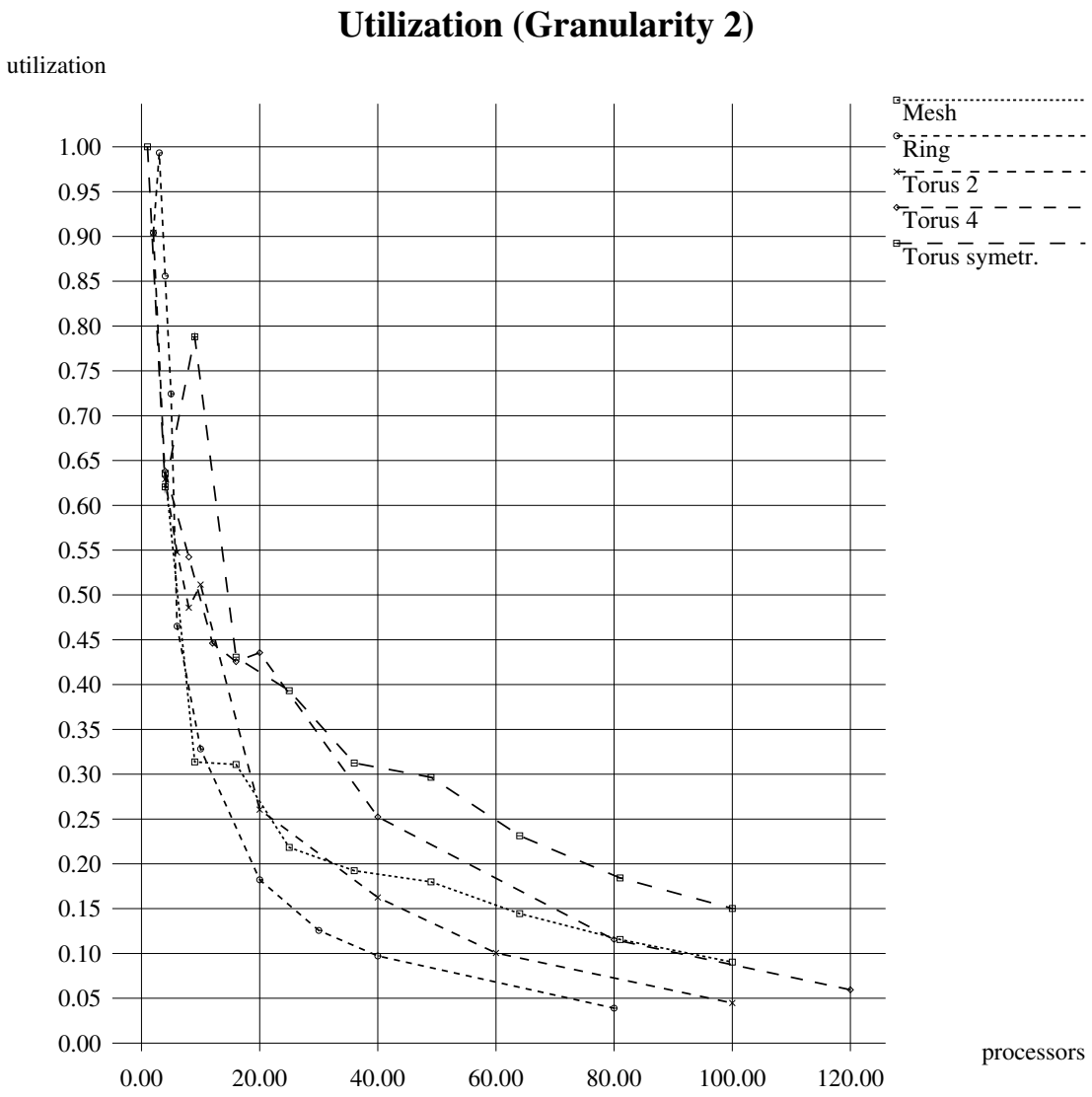


Figure 7.12: Utilization: $g = 2$, $n = 1000$

not obvious. All three seem to be equally good.

7.4 Estimating the quality of a topology

A way of comparing topologies without trying to schedule task graphs onto them would be very useful. In this section a single value characterizing a processor graph is proposed.

Two factors decide how big a maximum speedup can be:

- average distance between nodes (rather than diameter of the graph)
- number of parallel shortest paths between nodes

The first value is very easy to compute. It is formally defined as follows:

Definition 7.4.1 *Average distance, $\bar{d}$, of a processor graph $G_P = \langle V_P, E_P \rangle$ is a ratio of a sum of distances between all pairs of nodes to the number of such pairs.*

$$\bar{d} = \frac{\sum_{u,v \in V_P} a(u,v)}{|V_P|^2}$$

where $a(u,v)$ is the distance between u and v.

□.

Calculating the second one is in general not so straightforward, it can be, however, estimated by a number of links per node. Rather than taking the maximum degree, the average number of links per node (average degree, $\overline{\Delta}$) should be used. The average degree of a graph $G_T = \langle V_T, E_T \rangle$ is

$$\overline{\Delta} = \frac{|E_P|}{2|V_P|}$$

Average distance should be minimized, the average degree should be maximized, so a naive way of combining them is to use a ratio of one of them to the other. A "goodness factor" — a parameter characterizing a processor graph — can be defined.

Definition 7.4.2 *A goodness factor of a processor graph is a ratio of the average vertex degree of this graph to its average distance.*

$$\lambda = \frac{\overline{\Delta}}{\bar{d}}$$

□.

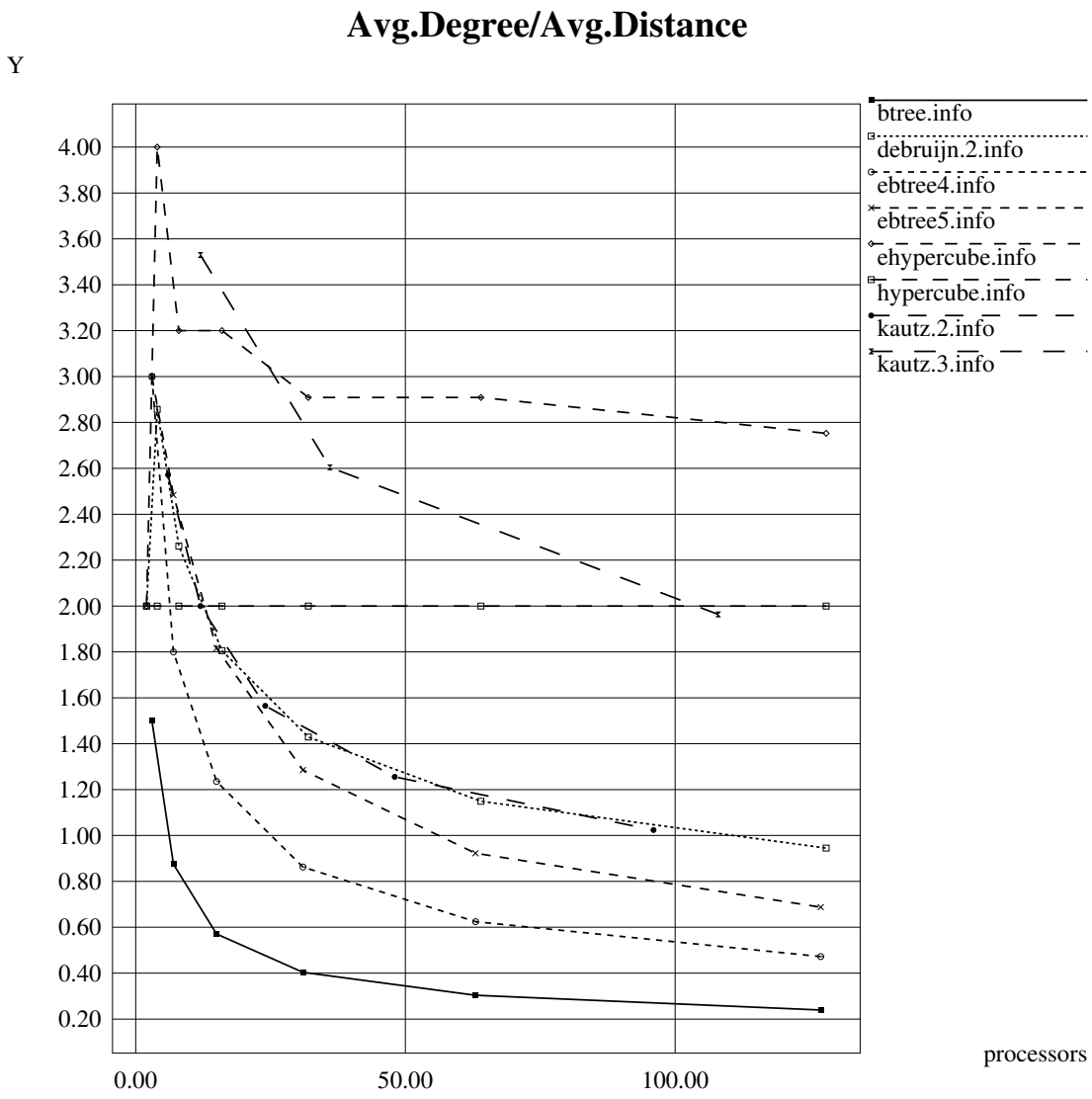


Figure 7.13: Goodness factors

Figures 7.13 and 7.14 show the graphs of a goodness factor. The correlation between speedups achieved in the third series of experiments, and the goodness factor is very big. The values for 64 processors are summarized in Table 7.2.

The relative qualities of T$(4, n)$ and ECBT5(n) and of CBT(n) and of R(n) are not estimated correctly. But taking into account that this is only a heuristic estimation of the quality of the topology, the correlation seems to be satisfactory.

Topology	Speedup (64proc.)	Goodness factor
Enhanced hypercube	25.5	2.9
Kautz graph UK$(3, n)$	21.0	2.3
Hypercube	19.0	2.0
Kautz graph UK$(2, n)$	16.0	1.1
Binary de Bruijn graph	15.0	1.1
Symmetrical torus	14.5	1.0
Torus T$(4, n)$	10.0	0.8
Enhanced binary tree ECBT5(n)	9.0	0.9
Symmetrical 2-D mesh	9.0	0.7
Enhanced binary tree ECBT4(n)	7.0	0.6
Torus T$(2, n)$	5.8	0.4
Complete binary tree	2.5	0.3
Ring	3.5	0.1

Table 7.2: Speedups and goodness factors for 64 processors

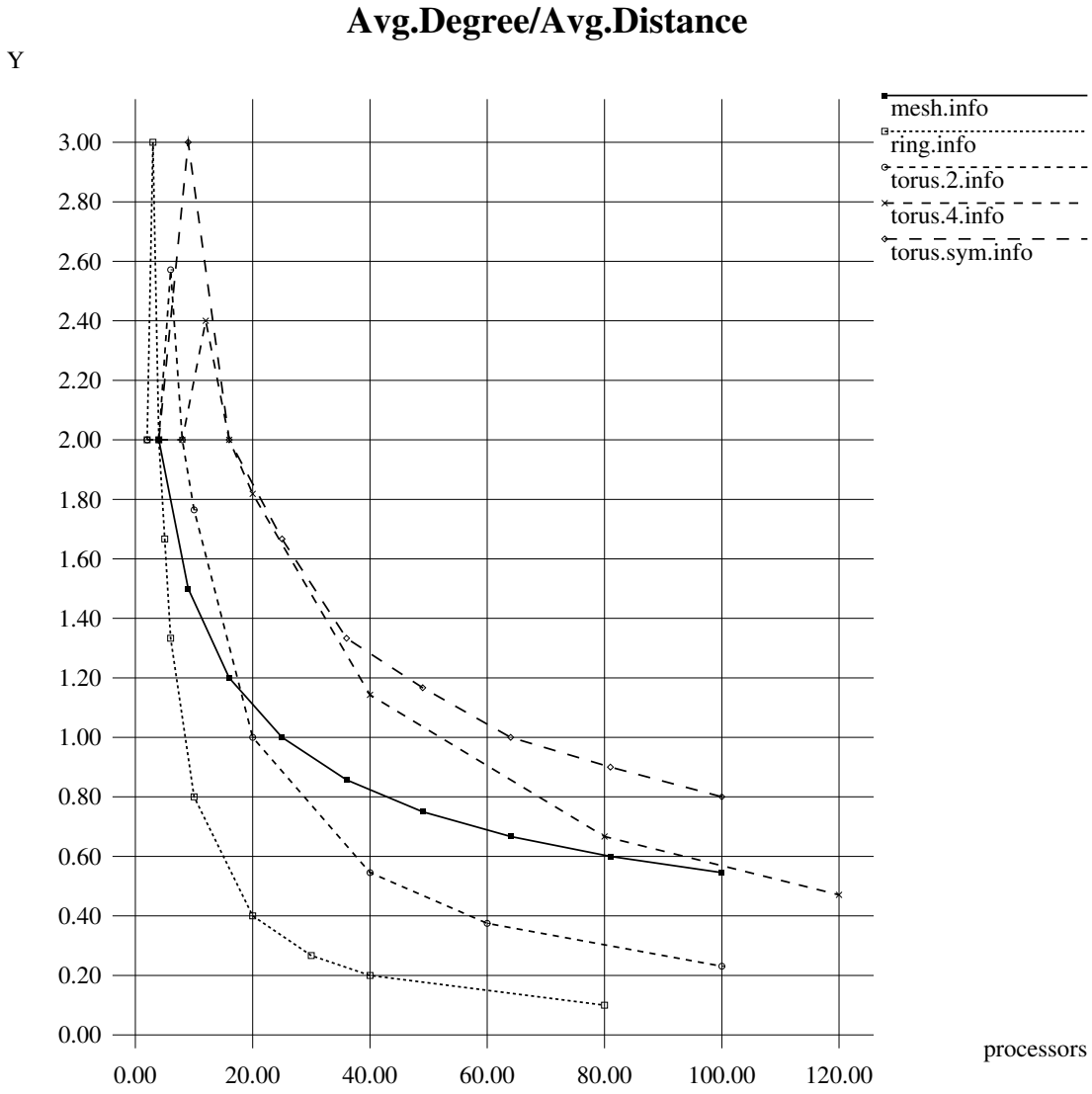


Figure 7.14: Goodness factors

Chapter 8

Conclusions/Outlook

The main results of this project are:

- *A new scheduling heuristic* **MMH**.
- *Comparison of several potential multiprocessor topologies.*
- *Proposal of a goodness factor as a way of estimating the quality of a topology.*

MMH is not a completely new heuristic. It is merely a modification of the already existing algorithm, the Mapping Heuristic. Schedules generated by **MMH** are in many cases identical to those generated by **MH**. A thorough comparison of the two heuristics has not been conducted. For a few processor graphs and task graphs which were considered, **MMH** performed favourably.

The main reason for modifying **MH** was, however, to obtain a faster algorithm. This goal was achieved — time complexity of **MMH** is lower than time complexity of **MH**. The **MMH** algorithm could be perhaps further improved. It seems unlikely that a lower time complexity could be achieved, but schedules of better quality could be possibly generated.

Comparison of several topologies revealed the ones which were best suited for this kind of applications, i.e. applications with random traffic. These results, of

course, do not preclude topologies which were evaluated here as "bad" from being used in other types of applications, e.g. programs showing so called geometric parallelism.

Further work is possible here by examining new topologies and by performing experiments for large task graphs obtained from real programs. Only small real-life task graphs were examined. Because numbers of processors were small, schedules for different multiprocessors were similar.

Lastly, further work is possible with the proposed goodness factor. Naively defined goodness factor showed a surprising correlation with achieved speedups. It seems possible, however, that a more accurate method for estimating a quality of a topology could be found by a more elaborate formulation.

The restricted amount of time made it impossible to investigate further these interesting problems.

Bibliography

[ACD74] Thomas L. Adam, K.M. Chandy, and J.R. Dickson. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Series in Coputer Science and Information Processing. Addison-Wesley Publishing Company, Reading, Massachusetts, April 1987.

[BDW86] Jacek Błażewicz, Mieczysław Dąbrowski, and Jan Węglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions On Computers*, C-35(5):389–393, May 1986.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From Control Flow to Dataflow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, June 1991.

[Bok81] Shahid H. Bokhari. On the Mapping Problem. *IEEE Transactions On Computers*, C-30(3):207–214, March 1981.

[BP89] J. C. Bermond and C. Peyrat. De Bruijn and Kautz Networks: a Competitior for a Hypercube? In F. Andre and J.P.Verjus, editors, *Hypercube and Distributed Computers*, pages 279–293. Elsevier Science Publishers, North-Holland, 1989. Proccedings of the First European Workshop, Rennes, France, 4–6 October 1989.

[Chr89] Philippe Chretienne. Task Scheduling over Distributed Memory Machines. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 165–176. Elsevier Science Publishers, North-Holland, 1989.

[CL86] Gary Chartrand and Linda Lesniak. *Graphs and Digraphs.* Mathematics Series. Wadsworth and Brooks/Cole Advanced Books and Software, Monterey, California, second edition, 1986.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1990.

[CSG89] Woei-Kae Chen, Matthias F. M. Stallmann, and Edward F. Gehringer. Hypercube Embedding Heuristics: An Evaluation. *International Journal of Parallel Programming*, 18(6):505–549, 1989.

[Dar91] Alain Darte. Two Heuristics for Task Scheduling. May 1991.

[Deo74] Narsingh Deo. *Graph Teory with Applications to Engineering and Computer Science.* Series in Automatic Computation. Prentice Hall, Englewood Cliffs, N.J., 1974.

[ERL90] Hesham El-Rewini and T. G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[ERS90] F. Ercal, J. Ramanujam, and P. Sadayappan. Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *Proc. 19th Design Automation Conference*, pages 175–181, June 1982.

[GIS77] Teofilo Gonzalez, Oscar H. Ibarra, and Sartaj Sahni. Bounds for LPT Schedules on Uniform Processors. *SIAM Journal on Computing*, 6(1):155–166, March 1977.

[GNP90] David Gelernter, Alexander Nicolau, and David Padua, editors. *Languages and Compilers for Parallel Computing*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, first edition, 1990.

[GZ87] John R. Gilbert and Earl Zmijewski. A Parallel Graph Partitioning Algorithm for a Message-Passing Multiprocessor. *International Journal of Parallel Programming*, 16(6):427–449, 1987.

[HCAL89] Jing-Jang Hwang, Yuan-Chien Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[HS88] Dorit S. Hochbaum and David B. Shmoys. A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach. *SIAM Journal on Computing*, 17(3):539–551, June 1988.

[KBG90] Andrew W. Kwan, Lubomir Bic, and Daniel D. Gajski. Improving Parallel Program Performance Using Critical Path Analysis. In Gelernter et al. [GNP90], pages 358–373.

[KL70] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49:291–307, February 1970.

[LA87] Soo-Young Lee and J. K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Transactions On Computers*, C-36(4):433–442, April 1987.

[MT91] Sathiamoorthy Manoharan and Peter Thanisch. Assigning Dependency Graphs onto Processor Networks. *Parallel Computing*, 17(1):63–73, April 1991.

[PW87] Shlomit S. Pinter and Yaron Wolfstahl. On Mapping Processes to Processors in Distributed Systems. *International Journal of Parallel Programming*, 16(1):1–15, 1987.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, first edition, 1989.

[Sco91a] Rob Scott. Intel Scientific Computers. In Trew and Wilson [TW91a], chapter 4.1, pages 126–136.

[Sco91b] Rob Scott. NCUBE Corporation. In Trew and Wilson [TW91a], chapter 4.2, pages 137–148.

[SH86] Vivek Sarkar and John Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. *SIGPLAN Notices*, 21(7):17–26, July 1986.

[SP89] Maheswara R. Samatham and Dhiraj K. Pradhan. The De Bruijn Multiprocessor Network: A Versatile Parallel Processing and Sorting Network for VLSI. *IEEE Transactions On Computers*, C-38(4):567–581, April 1989.

[SS88] Youcef Saad and Martin H. Schultz. Topological Properties of Hypercubes. *IEEE Transactions On Computers*, C-37(7):867–872, July 1988.

[TW91a] Arthur Trew and Greg Wilson, editors. *Past, Present, Parallel: A Survey of Available Parallel Computing Systems.* Springer-Verlag, Berlin, Heidelberg, first edition, 1991.

[TW91b] Nian-Feng Tzen and Sizheng Wei. Enhanced Hypercubes. *IEEE Transactions On Computers*, C-40(3):284–294, March 1991.