The Network of Tasks Model

D.B. Skillicorn
skill@cs.queensu.ca

May 1999
External Technical Report
ISSN-0836-0227-
1999-427

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared May 13, 1999

Abstract

We present a new model, the Networks of Tasks (NOT) model, that allows modules from skeleton-like languages to be embedded in static task graphs. The model is designed to provide transparent cost information, so that program designers can accurately predict the execution time performance of their programs as they assemble them. This is done by using an implementation technique called *work-based allocation* which uses adaptivity of the component node programs to execute the task graph with the same work and communication cost that is visible when the task graph is assembled. The semantics of NOT programs is simple enough that formal methods for developing them are straightforward. A refinement-based calculus for the NOT model is also outlined, and a law for handling residuals is given.

Keywords: task graph, adaptive computation, heterogeneous computation, program transformation, cost modelling, formal methods, refinement calculus.

1 Introduction

Constructing parallel software is made easier and more cost-effective if the parallel programming model (that is, the abstract machine to which the programming language is targeted) has the following properties:

- It is *portable* so that programs can be written once, but executed on many different platforms without further modification;
- It is *effective* at using the available resources of target platforms, so that performance justifies using parallelism; and
- It is *predictable* so that the costs (e.g. execution times) of programs can be computed more directly than by executing them.

Predictable costs are necessary if programs are to be designed, since without them programmers cannot make informed choices between different data representations and algorithms [15]. Without such costs, parallel programming remains a black art, and programs tend to be based on the first plausible design, rather than the best design.

Predictability for costs requires the programming model to have an associated cost model that is:

- *accurate* – predicted costs are close to those observed by executing programs;
- *simple* – only a small number of program and architectural parameters are required to compute costs;
- *monotonic in the architectural parameters* – when the machine parameters get better, programs get faster;
- *convex in the program parameters* – decreasing the cost of the piece of the program decreases the cost of the whole program.

Inaccuracy makes a cost model useless, and may even encourage poor designs by misrepresenting which parts of programs are expensive. Complexity makes costs inaccessible to designers, who will therefore not use them. Lack of monotonicity impacts portability, since a better machine will not necessarily run a particular program faster. Lack of convexity means that programs must be built as a whole, since a programmer cannot safely conclude that improving a piece of the program will necessarily improve the entire program.

Convexity is an easy property to achieve for cost models of sequential programs, as these tend to be additive: the cost of a program is the sum of the costs of its pieces. However, in a parallel setting, convexity is much more difficult. For example, congestion is a non-linear phenomenon of communication networks. Slowing down or postponing the execution of one part of a program may decrease applied load on the interconnect at a critical moment and increase the throughput seen by other parts. This may well improve the overall performance. A programmer can never be sure, in most parallel programming environments, whether it is a good idea to communicate as early as possible, or at some later time.

2 Existing Parallel Models

In the past decade, several families of parallel programming models have been developed that satisfy the requirements outlined in the previous section. They are all based on sequential composition of building blocks that fill the 'width' of the target platform. The use of sequential composition means that the cost of a program is once again a summation: the sum of the costs of the individual building blocks. For many of these models, the set of available building blocks is finite, and so the cost (and implementation) for each can be worked out in advance. Many such models also allow building blocks to be *adaptive*, either because they have accompanying transformation systems which allows for substitution of functional equals by different implementations; or because blocks can be internally rearranged to use, say, fewer processors.

Examples of families of models are:

1. Data-parallel languages, in which each processor executes the same instruction on different data associated with each virtual processor [9]. The cost of the parallel program is the same as that of the (sequential) program executed by each processor.

2. SPMD languages, which relax the precise instruction-by-instruction synchronisation of data-parallel languages in favour of a more-sporadic synchronisation. However, each processor executes the same (sequential) program and so the cost of the parallel program is larger than that of a single sequential program by a predictable overhead term describing the cost of the necessary synchronisations.

3. Skeleton languages [6], which provide a set of basic internally-parallel operations from which programs are constructed. There is wide variety of ways in which these basic operations are chosen, from the categorically-motivated choices of BMF [3], to pragmatic choice based on particular application domains [7, 13]. Most skeleton languages provide costs for each skeleton, and program costs are simple sums of these basic costs.

4. Models such as Bulk Synchronous Parallelism (BSP) [18] and Calypso [2] that use virtual barrier synchronisations to reduce parallel programs to sequential compositions of supersteps. The content of supersteps can be arbitrary, but their costs can be readily computed based on local computations and the volume of communication involving each processor.

These models have been quite successful in certain applications, but they clearly have some limits. In particular:

- it may be unnatural to express some computations as the single sequence of blocks that these models require;

- the (virtual) parallelism expressed in the program may be smaller than the parallel capacity of the target platform, so some processors are necessarily idle;

- the application may have dynamically-varying structure that depends on the values, not just the sizes, of the input data.

In each of these three situations, skeleton-like programming models are ineffective.

One step towards a solution is to relax the requirement that programs are *sequential* compositions of building blocks, and instead allow them to be directed graphs, whose nodes are blocks as before and whose edges represent the data dependencies that must now be made explicit. Such graphs are often called *task graphs*.

In this paper we define a task graph model, the Network of Tasks (NOT) model, explicitly designed for accurate cost modelling. This constrains choices substantially, but the resulting model is reasonably expressive. Because the model has a simple semantics, it is also possible to define a software development methodology for the model, based on extending the Refinement Calculus. In Section 3 we look at the properties of existing task graph models and languages that are similar to the NOT model. In Section 4 we introduce the NOT model in detail. In Section 5 we present several implementation techniques that produce a cost model with the desired properties. In Section 6 we describe a refinement system for the NOT model and discuss the issue of graph residuals.

3 The Design Space of Task Graphs

Task graphs are not a new idea. Dataflow programs are examples, although ones in which the graph nodes are single instructions. We outline the design choices that can be made in designing task graphs, and compare several existing designs for task graphs as parallel programming models.

Task graph models and languages tend either to be extensions of data-parallel languages, adding the capability to introduce tasks; or extensions of task-based languages in which the nodes become internally parallel. Models tend to reflect something of their origins in their design – in particular those of the first kind tend to have a distributed view of memory, while those of the second kind tend to have a shared view of memory.

Here are the design choices for task graphs:

a. How rich a structure is allowed for the graph?
b. What substructures are allowed in the graph?
c. Are cycles permitted, and under what circumstances?
d. What is the memory model?
e. What types of nodes are allowed?
f. Is state maintained between nodes?
g. Is the graph structure static or dynamic?
h. How are edges identified?
i. What is the semantics of edges?
j. What is the implementation strategy?
k. What is the cost model?

Three models of the first kind, that is extensions of data-parallel languages that enrich them with a graph structure, are COLT [12], SkIE [21], and Rauber and Rünger's Task Graphs [14]. Each of these models allows stateless nodes written in an imperative language, connected by edges representing variables that are implicitly distributed if the variables concerned are non-scalar, and pipelined. The set of graph substructures differs slightly: COLT allows pipelines and farms, while SkIE extends this with a seq and a loop construct. Task Graphs allow only seq and par constructs. COLT and SkIE use names to associate the heads and tails of communications, while Task Graphs use position as well. All three models use a global heuristic to optimise the distribution of data and the choice of implementation for each node.

Other models have developed from approaches that already incorporate the idea of a task graph. Bal and Haines [1] give a good survey. One well-known example is Opus [5]. Programs are collections of parallel objects that interact via *shared data abstractions* which manage the invocation of remote methods, synchronously or asynchronously. Objects are statically-allocated, but data are dynamically redistributed at calls. Fx [19, 20] is an extension of HPF in which subsets of data-parallel code can behave as if they were independent processes runnning on *processor groups*. This allows a monolithic data-parallel program to be decomposed into a hierarchy of smaller data-parallel programs with a graph structure connecting them. The connections have copy to and from semantics.

4 The Network of Tasks Model

The Network of Tasks (NOT) model extends distributed-memory approaches such as COLT and SkIE to richer graph structures and uses implementation techniques that permit simple cost models.

A NOT program is an arbitrary directed graph, acyclic except within certain substructures. Arbitrary subgraphs may be encapsulated as named objects, but the only effect is to provide a scope for edge names within the subgraph. There is also a loop construct of the form:

```
first
    block1
while predicate on edge values
    block2
```

Both *block1* and *block2* may be arbitrary blocks, but *block2* has the same edge names as its inputs and outputs. Edge names in *block2* that appear outside the loop construct get the values produced by the last execution of *block2*, the one for which the predicate is false. Note that *block2* is executed repeatedly in its entirety, that is the repetition is at the level of blocks, not within *block2*. The loop construct is semantically equivalent to a path in the task graph beginning with *block1*, and continuing with a sequence of *block2*'s whose length is not known at compile time, but whose other properties are. Graph structures are static, apart from the dynamic behaviour of the loop construct.

Nodes in a NOT graph are arbitrary programs from models discussed in Section 2, and may be heterogeneous. Each node program must be minimally adaptive, in the sense that Brent's theorem can be used to reduce its parallelism at the expense of its execution time, and must provide a cost expression parameterised by the number of processors on which it executes. Nodes may also describe any deeper adaptivity that they possess, and this may be used by the NOT implementation system.

The NOT model assumes a distributed-memory implementation, and no state is retained between blocks. Programs in the model may be compiled to execute on a single parallel computer, or may execute on an ensemble of different computers (although it is not intended for the sort of wide-area computation exemplified by the Globus project [8]).

Edges are identified by name, using a single assignment semantics in a textual form of the graph, or as edges in a visual form of the graph. The data that flows along an edge is a list of variable values, where the variable may be scalar or structured. For simplicity, a set of edges may be regarded as a single edge, with a bag of lists of values flowing along it.

There are three types of edges distinguished by the timing of data flow:

File of – The whole list of values must exist before any transfer of data takes place. This is the basic type of edge. The name of the edge may be a simple name, in which case the output of one node is connected to another directly. The name may also be a file name when it appears as a node output, and a URL when it appears as a node input. This allows intermediate values to remain in existence, as files, after a program has terminated; and also allows single programs to be distributed over ensembles of computers.

Block of – Data is pipelined in blocks of instances, determined either by the compiler or by the programmer.

Bag of – The ordering of the data is not important, and it may be generated and consumed in any order (but is available in a pipelined fashion).

Thus a typical program might look like:

```
graph a in(x,y : file of int(100, 100)) out(t: file of real(100))
   prog1 in(x) out(z: file of real(100))
   prog2 in(y) out(q: file of real(10))
   prog3 in(z,q) out(t)
end
```

The NOT model does not really address programs whose behaviour is dynamic, since the task graph structure itself is static, except for the loop construct. There is always a difficult trade-off between dynamic behaviour and cost modelling, since complex decisions at run-time are hard to model accurately at design time.

5 Implementation

The implementation of the NOT model is motivated by the need for a useful cost model as well as by concerns with portability and effectiveness.

Let us begin by defining some terms. Consider the sets of nodes in the graph that are disjoint in the sense that there is no path connecting any pair of them. Such sets represent nodes that might be simultaneously active in some execution of the graph. The *graph width* is the cardinality of the largest such set. The length of a task graph is the number of layers (that is, the number of nodes) along the longest path from source to sink.

Nodes may be adaptive in two senses. There may be different implementations that capture the functional behaviour of a node. These different implementations often have different costs, particularly for communication. The NOT model makes these implementation choices to globally optimise the task graph cost.

The second kind of adaptivity is the tradeoff between processor resources and execution time of the sort described by Brent's theorem [4]. The nodes of a task graph have a cost associated with them, expressed in terms of the number of processors available to execute that node. It is useful to think of these as the vertical (time) cost of the node, given a (horizontal) number of processors used, and natural to write such nodes in a way that captures the maximum available parallelism. This is because Brent's theorem gives a generic way to execute a parallel program on fewer processors than it was designed for, with a corresponding increase in execution time, so that the total work (product of time taken and processor used) remains constant. Each set of disjoint graph nodes has a width based on the sum of the (virtual) parallelism of the nodes it contains. The *total width* of a task graph is the largest such sum.

The implementation strategy for a task graph depends on the relationship between the width of the graph (which corresponds to the available virtual parallelism) and the number of the processors in the target, p (which is the available physical parallelism).

Roughly speaking, there are three situations:

- If $p <$ graph width then there is little point in trying to exploit intra-node parallelism, and not even the parallelism of the task graph itself can be fully exploited.
- If graph width $< p <$ total width, then the parallelism of the task graph can be fully exploited, and there may be some opportunity to exploit intra-node parallelism as well. In this case, we use *work-based allocation* as the implementation technique.
- If total width $\times$ length $< p$ then it becomes sensible to use pipelining to execute all nodes of the graph simultaneously. It may also be possible to use farming to increase the effective parallelism of single nodes in the graph, or indeed of the entire graph.

In fact, as we shall see, it is not the total width that is important so much as the effective total width, which is usually smaller and depends on how much slack there is to move computation forward or backward in the graph, effectively reducing the parallelism needed at particular times.

Of course, these are general guidelines. There are programs in which a single node forms a bottleneck for which a farm may be useful; and programs with a path on which much of the computation lies for which pipelining may be useful.

5.1 Implementation for Small p

Increasingly, systems with modest parallelism ($p \leq 4$) are becoming available, but techniques for simply and effectively programming them are lagging behind. Multithreading, for example, requires relatively sophisticated understanding of synchronisation and memory management. There is some attraction to implementing the NOT model in such a way that it make effective use of small-scale parallel systems as well as those with greater parallelism.

Task graphs express two levels of parallelism: that of the graph itself, and that of the nodes. When the physical parallelism is small, the overheads of communication dominate. In this setting, it seems sensible to execute the nodes sequentially, and use the limited parallel capacity at the level of the graph.

Optimal scheduling of a graph of nodes representing sequential computations is a well-studied problem [11], and algorithms are typically exponential in the number of nodes in the graph. In practice, heuristics are used. When the problem is to schedule a graph on a small number of processors, simple heuristics exist. For example, there are graph drawing algorithms that produce good layouts for DAGs in time linear in the number of nodes. Such drawings become schedules by drawing lines through the layout, and using the regions so produced as processor allocations.

5.2 Implementation for Moderate p

For larger target architectures, it becomes appropriate to exploit both the parallelism within the task graph nodes, and the parallelism between graph nodes. For this, we use *work-based allocation*, a technique that exploits inter-node parallelism first, then intra-node parallelism, and does so in such a way that performance is predictable.

We consider first the implementation of task graphs without the loop construct. Every such graph can be mapped to a series-parallel structure in which the work requirements for each subgraph can be easily seen. This does not always produce an optimal schedule, but it produces one whose structure is readily visible at the software level.

When different implementations of each node are possible, finding the choices which optimise both total computation and total communication is difficult. Many approaches to modelling the cost of communication have been tried. Typically, these are based on a transmission cost per byte, and perhaps an overhead term for each message. It is hard to account for the effect of congestion on message transit times because it is a global property that depends on the total applied load. The BSP cost model [18] has been shown to be accurate over a wide range of real computers and application programs. In this model, the cost of communication depends on the volume of communication into and out of each processor. Each parallel computer is characterised by a single parameter, g (in units of instruction execution times per byte), which measures the permeability of the interconnect to continuous random traffic. A node that sends and receives a total of h bytes will be charged hg (NB in units of instruction execution times) for its communication. The total communication cost of a set of processors communicating is the *maximum* of these hg terms. All of the congestion-like properties are captured in the measurement of g which can be regarded as the reciprocal of a resistance. In the true BSP model, communication takes place, in all processors, at the end of supersteps, that is roughly simultaneously. However, work-based allocation creates a similar situation, since the computation phases of nodes will tend to complete at similar times. Therefore, the BSP model for communication is likely to be accurate.

When the variable associated with an edge of the task graph is non-scalar, it will typically be computed by code that is distributed over the set of processors executing its producer node, and be destined for the set of processors executing its consumer node. Thus each graph edge is in fact a bundle of edges from a set of processors on which the source node is executing to a set of processors on which the destination node is executing. If the nodes of a task graph are explicit programs written in some language such as HPF then the distribution of variables is completely determined by the program, and the exact pattern of data movement along a bundle of edges is completely determined by this distribution. However, when the programming language in which the node program is written is based on a model that allows transformations between semantically-equivalent textual forms, these different forms may have different distributions for variables that are moved along graph edges. The best form for a node program can only be computed by including the cost of communication, including redistributing input and output data structures. This requires (at least) a cost model for communication along graph edges.

This problem can be solved by using the BSP cost model for communication and a shortest-path algorithm for searching alternate implementation structures. The paper [17] shows how to solve the global optimisation problem for the case where the parallel program structure is a sequence of nodes, each of which might have multiple parallel implementations, and communication is costed using the BSP cost model. The algorithm produces the optimum in time polynomial in both the number of nodes in the sequence, and the number of alternatives for each step. This algorithm can be extended to the graph case.

The basic algorithm works by finding the shortest path through a graph whose nodes are alternate implementations of each step, labelled with their costs, and whose edges are communication patterns necessary to connect different implementations, also labelled with their costs. Using the BSP cost model means that these costs are in the same units, reducing the problem to one-dimensional minimisation.

The graph version of the algorithm works, layer by layer, through the graph. Suppose for simplicity that every one of the n layers of the graph consists of w tasks, and that there are a alternative implementations of each task. Then there are a^w possible configurations of the first layer. Each of these configurations can be extended to the next layer by choosing one of the a^w possible configurations for the next layer. Thus the total cost for the algorithm is $a^{2w}n$. This is likely to be a very pessimistic bound in practice. The value of a is likely to be at most 2 or 3, with many tasks only having one plausible implementation.

Task graphs whose nodes are adaptive in this strong sense of possessing multiple implementations for nodes have now been transformed to a form that is optimal with regard to these choices. It now remains to transform the task graph as a whole to best use the available processors.

If all nodes of the graph are regarded as identical, then the graph can be arranged in layers, beginning with those nodes whose input edges come from outside the graph, then those nodes whose inputs come only from nodes in the first layer, and so on. This arrangement defines the earliest layer for each graph node. Repeating the procedure from the end of the graph gives an arrangement in which the layer defined is the latest layer for each graph node. Each node is now labelled with its earliest and latest layer.

A node that spans i layers is divided logically into i sequential nodes, each of the original width and taking an ith fraction of the execution time of the original node. The graph now consists of nodes in layers. The schedule aims to have each layer complete at the same time.

The node in each layer that takes the longest to complete is the one with the largest

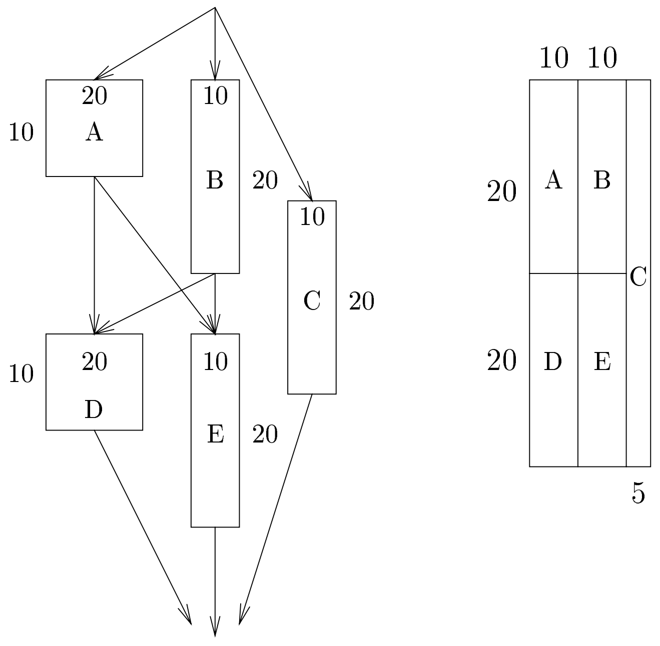

Figure 1: Task Graph and its Rectangle Form

execution time, and nothing can be done to speed it up because it has been expressed with maximum available parallelism. Instead, all of the other nodes in the layer are slowed down, by allocating fewer processors to them than they can take advantage of, so that each finishes at about the same time as the slowest. (This will not work out exactly because of rounding errors.) If the slowest node in a layer takes time t' then a node with costs p and t will be reallocated to

$$p' = \frac{pt}{t'}$$

processors.

The allocation of processors to nodes using this algorithm allows each to complete at the same time as all of the other nodes whose latest layer in the same as its own. In other words, the use of p' processors causes the graph to be executed in real time in the same relative order as in virtual time. Nodes with a large slack (the difference between earliest and latest layers) tend to be assigned few processors, while those with small slack tend to be assigned many processors. The work cost of the task graph is the same as the sequential cost (that is, perfect speedup) as long as no more processors are used than the effective maximum width of the graph.

An example is shown in Figure 1. Each node in the task graph on the left is labelled with its maximal parallelism and time required for execution (that is, the length of its critical path). The graph width of this task graph is 3, and the total graph width is 40. The total amount of computation required is 1000, which is the time that the entire graph would take to execute sequentially. The rectangle form of the graph is shown on the right hand side of the figure. Nodes A and B are both in layer 1, node C spans layers 1 and 2, and nodes D and E are both in layer 2. From this it can be seen that the maximum effective width of the graph is 25, and its total execution time using this number of processors is 40. Allocating any further processors does not reduce the execution time. Brent's theorem can be applied straightforwardly to the rectangle, so that, for example, 10 processors would be allocated 4,4,2 to the three subrectangles, giving a completion time of 100.

Of course, naively applying this scheme may require very different numbers of processors in different layers. It is straightforward, however, to incorporate a constraint on the number of available processors into the algorithm. (Of course, a layer with very little available parallelism may still underutilise the available processors – but this is unavoidable, and visible to the programmer.)

In theory, the overall work of the work-based allocation scheme is the same as the total work of the component nodes, and the total actual execution time is the total work divided by the number of actual processors available for allocation. This may not work out exactly because the number of processors allocated to a node must be an integer, and so there will be inaccuracies because of roundoff. There may also be inaccuracies because of nodes that span multiple layers. Such nodes are always spread over time, but it may happen that they could have been executed by spare processors in a particular single layer. Even in the pathological case, the error is at most a factor of two.

It remains to extend work-based allocation to include the costs of communication. The reason that this allocation scheme works for computation is that (approximately at least) the time taken for the computation part is given by

$$\text{time} = \text{work/processors}$$

In other words, as the number of processors is reduced, the time taken grows linearly. The BSP cost model for communication has the same property. If the volume of communication into and out of a processor is h for a program with virtual parallelism p, then the communication cost of a program implemented on p' processors is roughly

$$\text{communication cost} = h/(p/p')$$

Work-based allocation, however, allocates the number of processors for the upstream and downstream sides of a communication pattern independently. Fortunately, the same scaling property applies to the fan-out and fan-in independently. The cost of a communication pattern can be divided into these two costs, the fanout cost allocated to the upstream node, and the fanin cost allocated to the downstream node. The effect is that nodes that involve a lot of computation, or have heavy communication needs will be allocated more processors, preserving the layered execution property of work-based allocation.

Separating the input and output cost of communication does introduce a slight problem. Consider a communication step in which one processor sends messages to p others, and one processor receives p messages, one from each of the others. The standard BSP cost model charges pg for such a communication pattern. By separating the cost into fanout and fanin terms, we increase the apparent communication cost. Thus we must be careful to separate the costs used as the basis for processor allocation from costs that might be reported to the programmer, reflecting the performance seen at execution. (In any case, the BSP cost model's choice to aggregate the fanout and fanin costs is somewhat arbitrary, and other variants have been suggested.)

This scheduling algorithm does not handle loop constructs, since it is static and the number of iterations of a loop body cannot be known at compile time. The width of the body node is known, and indeed we only need to choose a communication pattern once since a loop is a path connecting identical nodes. However, we do not know the total work that the loop will do, so we do not know how best to allocate processors to it. There are two approaches to this. The first is to use a static estimate of the number of times each loop will iterate (a compiler writer's rule of thumb is that this is 10). The second is to handle processor allocation at runtime, rather than during compilation. This has the added advantage that it allows other dynamic behaviour in the task graph such as node spawning, provided that, as before, a cost expression is available for each node. The difficulty with dynamic scheduling is that there is no way of knowing, in advance, which processors will execute which nodes, so that code must be loaded into many more processors than actually need it.

5.3 Pipelining

Pipelining can improve the performance of a task graph if there are more processors available than can be used by the effective width of the graph. In effect, pipelining allows the depth of the task graph to be used as well. There are two requirements, however, for pipelining to be practical:

1. The modules at the nodes of the task graph must have semantics that allows them to work on blocks of data at a time. Roughly speaking, this means that modules must have an outer loop that can be unrolled into blocks of different sizes. For several of the models we have discussed, implementations of this kind are possible.

2. There must be enough processors available to allow the entire task graph to be pipelined, since pipelining the first few stages and then reverting to nonpipelined for the remainder does not provide any overall performance improvement.

It is reasonable to expect that the cost of a node module becomes

$$\text{pipelined cost} \;=\; \text{cost}/(\text{block size}/\text{file size})$$

given the semantics above.

Work-based allocation can still be used to schedule processors for pipelined execution. Notice that work-based allocation has the property that each 'layer' of the task graph finished at approximately the same time, by construction. If the same relative allocation of processors is made, this means that the pipelined versions of each layer will have the same behaviour as the nonpipelined versions. In other words, the modules in each layer will produce their next block of outputs at about the same time. Logically speaking, pipelined work-based allocation allocates processors in the rectangle version of the task graph described previously using the *volume* of each rectangle. However, since the lengths of nodes being considered simultaneously are the same, this is equivalent to allocating them based on effective width. Thus we get the same allocation as in the non-pipelined case.

It is less clear that communication cost modelling remains the same. In our earlier use of the BSP cost model we ignored any costs due to start-up effects (BSP's l term). This is not unreasonable when file of semantics are used for edges, since the total amount of data moved is large and start-up effects accordingly small. In the pipelined setting, the blocks actually transmitted may be quite small, and start-up effects significant. The cost of these extra effects need not affect the allocation used, since every message presumably pays the same penalty on start-up. They will, however, add to the overall cost of the program which will see a cost of

$$\text{number of blocks} \times \text{startup penalty}$$

between each pair of layers.

5.4 Farming

In a sequential composition of modules, farming can be used to balance a pipeline by introducing parallelism into the slowest module. In our task graph setting, we can instead vary the performance of a module by increasing or decreasing the number of physical processors allocated to execute it. However, there is still one situation in which farming can be useful, and that is to increase the *virtual* parallelism of a module. Using a farm allows us to get around the limitation of Brent's theorem in a situation where we would like to allocate *more* physical processors to a module than its virtual parallelism allows, in order to shorten the execution time of the layer in which it appears. Once again, there are two necessary conditions: the semantics of the module must allows its input and output edges to have bag of semantics; and there must be extra physical processors that could not otherwise be effectively used.

There is one other situation in which farming is commonly used: when the cost of processing a block of input varies significantly with the *values* of that input. At the level of cost modelling we have been assuming, such cases are not easy to extract since we have assumed cost expressions that are upper bounds, determinable at compile-time. This use of farming can be exploited in systems that use exact cost expressions and minimise over every element of a data stream, but cannot naturally be handled by the sorts of cost model we have been advocating. There is one exception: it is possible that when a data stream is unpacked from file into blocks, the terms of the nested cost expression may have different sizes. A farmed implementation would enable a maximum (for an upper bound) to be replaced by an average if the semantics of the module allowed it.

6 Software Development

Very little is known about formal software development in the context of graphs. When research into dataflow graphs was active, there was not enough interaction with formal methods researchers to stimulate cross-fertilization.

6.1 Extending the Refinement Calculus to Graphs

The Refinement Calculus [10] is a set of rules for manipulating specifications and sequential programs. Specifications are of the form

$$\text{frame} : [\text{precondition}, \text{postcondition}]$$

where the *frame* is a list of the variables that may be altered and the precondition and postcondition are predicates on these and other variables. A specification is an abstract computation that, started in a state satisfying the precondition, terminates in a state satisfying the postcondition, altering only variables mentioned in the frame. The calculus is a set of laws that define when one specification refines another. Laws are justified by weakest precondition reasoning, and so the Refinement Calculus can be viewed as a packaged form of weakest precondition software development. Some specifications are designated as *code*, that is specifications that are directly executable, based on the target environment chosen for the calculus. The Refinement Calculus is used to transform (by refinement) a high-level specification into an executable program. It is important to note that the calculus itself does not automate any of a derivation. It is possible to derive infeasible programs, even when feasible ones exist. Rather, the role of the calculus is to reduce the search space of implementations, guide the developer to regions that are likely to contain good software solutions, and provide a record of the design decisions made.

The Refinement Calculus assumes a single, global memory with unlimited read access, so it is not directly applicable to task graphs. However, the style of the Refinement Calculus is useful in developing a new calculus that does apply to graphs. A similar extension has been used to define a calculus for BSP programs, for instance [16].

Because the NOT model is based on distributed memory, and no state is preserved between blocks, it is straighforward to construct a refinement calculus for task graphs (RC-NOT).

Definition A *graph specification* has the form:

$$\text{inputs}, \text{outputs} : [\text{pre}, \text{post}]$$

where inputs and outputs are bags of lists of variable names in which the names appearing in inputs are unique, pre is a predicate mentioning only names appearing in inputs, and post is a predicate.

A graph specification describes an abstract computation that, given values for the variables appearing in inputs satisfying pre, terminates and produces values for the variables in outputs satisfying post. A graph specification is *feasible* if there is a real computation with these properties.

The basic graph refinement law is this:

Law[Graph Refinement] Given a graph specification

$$\text{inputs}, \text{outputs} : [\text{pre}, \text{post}]$$

a graph refinement is:

- a partition of inputs into a bag of lists, and a partition of outputs into a bag of lists in which each variable name appears exactly once;

- a set, A, of graph specifications of the form

$$\mathrm{in}_a, \mathrm{out}_a : [\mathrm{pre}_a, \mathrm{post}_a]$$

  ($a \in A$) connected by a set of directed edges, e, where a directed edge is a list of variables whose *tail* is a specification in which e appears as an element of out_a or a bag in the partition of inputs; and whose *head* is a specification in which e appears as an element of in_a or as a bag of the partition of $\mathrm{outputs}$.

A predicate is associated with each edge in the following way. Any postcondition of a specification whose output bag consists of i lists can be expressed in the form

$$\bigwedge pred_i \wedge pred$$

where $pred_i$ mentions only variables in the ith list, and *pred* is any other predicate that does not contain a conjunctive subterm mentioning only variables from a single output list (so the $pred_i$ are as large as possible). The predicate $pred_i$ is associated with the edge corresponding to list i. In a similar way, the partition of the predicate pre produces associated predicates for the edges of inputs.

The set, A, of graph specifications must have the following properties:

- the directed edges do not form a cycle;
- the precondition, pre_a, of each specification is a conjunction of the predicates associated with the edges for which that specification is the head;
- the conjunction of the predicates associated with each edge in $\mathrm{outputs}$ implies post.

This complicated law expresses the fact that a specification can be decomposed into a directed graph of subspecifications, one for each module, and that the preconditions for each module are whatever is known about the values of its inputs, which in turn are determined by postconditions of upstream modules. Note that a module cannot know anything about the relationships of its inputs to other variables whose values it is not sent. This seems realistic for the kinds of applications we envisage, especially as we view modules as being independently-derived in some other formalism. It seems to be possible to weaken this restriction to allow other parts of postconditions to be associated with edges but this introduces messy scope issues.

Once again, the calculus provides guidance, but does not prevent false steps in derivations. For example, it is probably a good idea to ensure that *pred* in postconditions is the simple predicate *true* (so that nothing is computed that isn't used), but this is not required by the calculus.

The following law allows the loop construct to be introduced into a task graph.
Law[Loop introduction] A graph specification

$$\mathrm{inputs}, \mathrm{outputs} : [\mathrm{pre}, \mathrm{post}]$$

is refined by

```
first
    $\text{inputs}, \text{out} : [\text{pre}, \text{inv}]$
while $G$
    $\text{out}, \text{out} : [\text{inv} \wedge G, \text{inv} \wedge 0 \leq \text{varnt} < \text{varnt}_0]$
```

where:

> out is a bag of variables that includes output,
> G is a predicate on the variables in out,
> inv is a predicate such that $inv \wedge \text{not } G \rightarrow \text{post}$,
> varnt is an expression whose value is obtained by substituting values of variables after *block2* has executed, while varnt_0 is the same expression with the values of the variables as they were before *block2* executes.

The predicate G is usually called the guard and determines when the loop terminates. The expression varnt is usually called the variant and ensures that progress is made by each execution of *block2*.
The loop introduction law assumes file of semantics for edges. Other extensions are possible: loops on streams are integral to the language Lucid for example.

A number of other laws for graph specifications are required, but they are both technical and straightforward. For example, both *weaken precondition* and *strengthen postcondition* are necessary.

6.2 Graph Residuals

Software engineers have long hoped for reusable components from which programs could be built. Progress has been made with the increasing use of object brokers, but Java has perhaps made reusable components a reality in a new way. Of course, in a sense the libraries of numerical analysis libraries have meant that much scientific programming has reused code for many decades. In a formal setting, the presence of existing code units means that the calculus must have some way of capturing the *difference* between a given specification and the pre-existing module, that is capturing what remains to be done. This is usually called a residual.

In the NOT setting, what we want to know is: given a specification and an existing module, what graph structure is required to build a complete computation that satisfies the specification and uses the existing module. Suppose that we have an existing module A. Then any graph in which it is used has the structure shown in Figure 2. Note that there must be at least one edge between regions B and C, since otherwise the graph would contain a cycle (into and out of A). Given a specification for the entire graph, it is straighforward to derive the specifications for regions B and C. If the specification of A is

$$\text{in}_A, \text{out}_A : [\text{pre}_A, \text{post}_A]$$

and the overall specification is

$$\text{inputs}, \text{outputs} : [\text{pre}, \text{post}]$$

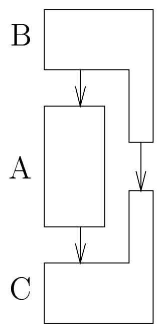

Figure 2: Residual of a given module A

then the specification of B is

$$\text{inputs}, \{\text{in}_A, \text{in}_C\} : [\text{pre}, \text{pre}_A \wedge \text{mid}]$$

for an arbitrary predicate, mid, and the specification of C is

$$\{\text{out}_A, \text{in}_C\}, \text{outputs} : [\text{post}_A \wedge \text{mid}, \text{post}]$$

Note that post_A and mid must be variable disjoint, while pre_A and mid may share variable names.

7 Discussion

The NOT model is a task graph extension for nodes that may be written in any skeleton-like programming language provided that a cost expression parameterised by the number of processors required can be generated. This allows it to be used to glue together heterogeneous programs written in skeleton languages such as BMF [3], P3L [7], BSP [18], or even HPF.

Unlike other task graph extensions, NOT emphasises performance transparency so that software developers can reliably predict the performance of their programs from the performance of the nodes and the graph structure. This is achieved by using an implementation strategy that preserves the work (product of processors and time) of the underlying node programs, and uses the BSP cost model to model to cost of communication.

The relatively simple structure of NOT programs makes it possible to give a refinement calculus style set of construction rules by which NOT programs may be derived from specifications. Unlike the sequential case, the application of a single rule produces not a single statement but the entire high-level structure of the NOT program.

References

[1] H.E. Bal and M. Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6, No. 3:74–84, July-August 1998.

[2] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, 1995.

[3] R.S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, February 1989.

[4] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21, No.2:201–206, April 1974.

[5] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. Technical Report 94-18, NASA Institute for Computer Applications ICASE, March 1994.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[7] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer-Verlag, August 1997.

[8] I. Foster and C. Kesselman. The Globus project: A status report. In *IPPS/SPDP'99 Workshop on Heterogeneous Computing*, pages 4–18, April 1998.

[9] P.J. Hatcher. The joy of data-parallel programming. In D.B. Johnson, F. Makedon, and P. Metaxas, editors, *Proceedings of the Dartmouth Institute for Advanced Graduate Study in Parallel Computation Symposium*, pages 19–30, June 1992.

[10] C. Morgan. *Programming from Specifications.* Prentice-Hall International, 1990.

[11] M. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 5(3):263–302, 1993.

[12] S. Orlando and R. Perego. Colt$_{HPF}$: A runtime support for the high level coordination of HPF tasks. *Concurrency: Practice and Experience*, to appear.

[13] S. Pelagatti. *Structured development of parallel programs.* Taylor&Francis, London, 1997.

[14] T. Rauber and G. Rünger. A coordination language for mixed task and data parallel programs. In *Proceedings of ACM Symposium on Applied Computing SAC'99*, Feb/Mar 1999.

[15] D.B. Skillicorn. Architectures, costs, and transformations. In *Constructive Methods for Parallel Programming, CMPP'98 (associated with Mathematics of Program Construction*, pages 1–15, June 1998. Fakultät für Mathematik und Informatik, Universität Passau, Report MPI-9805.

[16] D.B. Skillicorn. Building BSP programs using the Refinement Calculus. In *Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98)*, Springer Lecture Notes in Computer Science 1388, pages 790–795, March/April 1998.

[17] D.B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising data-parallel programs using the BSP cost model. In *Europar'98*, pages 698–703, September 1998.

[18] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[19] J. Subhlok, D. O'Hallaron, and T. Gross. Task parallel programming in Fx. Technical Report CMU-CS-94-112, School of Computer Science, Carnegie Mellon University, January 1994.

[20] J. Subhlok and B. Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, Las Vegas, NV, June 1997.

[21] M. Vanneschi. PQE2000: HPC tools for industrial applications. *IEEE Concurrency*, 6, No. 4, October-December 1998.