Chapter 1

Developing High-Level Irregularly-Parallel Programs for Multiple Architectures

M. KH. Aswad, H.-W. Loidl and P.W. Trinder[1]

Abstract: The variety of parallel architectures nowadays available implies that models for parallel software development should deliver acceptable performance on a range of architectures with minimal development effort. One approach is to use languages with very high level, and substantially architecture-independent, specification of parallel coordination.
This paper presents a *novel profiling-based methodology* for developing irregularly-parallel applications for multiple architectures in languages with high-level coordination. The methodology has two main phases: an architecture-independent phase of idealised parallelisation, and an architecture-dependent phase of accurate performance prediction.
The methodology has been distilled from the development of a dozen applications on various parallel machines. We have previously achieved good speedups with only minimal code changes: a speedup of 12.0 on a 16-processor workstation cluster and 2.8 on a 4-processor Sun SMP for a data-intensive program, and 11.9 on a 16-processor cluster for a symbolic computation application.
The methodology is illustrated by developing a substantial application for two very different architectures: a Beowulf cluster and a Sun SMP. Seven alternative parallel versions of the program are developed and evaluated using a simulated idealised architecture. Realistic simulation of the two target architectures accurately predicts the version that delivers the best performance. While the speedups achieved for this program are modest, 7.5 on a 30-processor Beowulf and 1.8 on

[1]School of Mathematics and Computer Science, Heriot-Watt University, Edinburgh, UK; Phone: +44 (0)131-451-3435; Fax: +44 (0)131-451-3732; Email: `{ceemka,hwloidl,trinder}@macs.hw.ac.uk`

a 4-processor Sun SMP, they require only minimal changes to the optimised sequential code and do not require any source code changes in porting the program between architectures.

1.1 INTRODUCTION

The performance of many parallel programs is very sensitive to the target parallel architecture, hence the development of parallel programs is typically highly architecture-dependent [ST98]. The goal of architecture-independent parallelism is to develop programs that can be migrated from architecture to architecture without sacrificing the efficiency or requiring redevelopment [CC00].

High level languages are one means of achieving architecture independence as parallel coordination is specified at a higher level of abstraction, i.e. with less reference to a specific underlying machine. Some languages encourage the specification of very fine-grained parallelism with largely implicit communication and rely on dynamic management of the available parallelism to achieve efficient parallel execution on a wide range of machines. Examples for this approach are Filaments [LFA96], Cilk [BJK+95], and Cid [Nik94] as extensions of imperative languages and Charm [KRSG94] as an extension of an object-oriented language. Skeleton-based languages use a set of higher-order functions with efficient implementations to achieve architecture-independent parallelism, e.g. PMLS [MSBK01] or P3L [BDO+95]. Data parallel languages like HPF [KLSS94], or NESL [Ble93] restrict their scope to one paradigm and use the additional information on the dynamic structure of the computation to tune the execution for a particular parallel architecture.

Programs in high-level parallel languages are typically tuned to a new architecture using a combination of dynamically-adapting implementation and static/ dynamic analyses, i.e. cost models or profiling. Modifying the program using cost models or profiling may be automatic, but is in most cases left to the programmer. Static cost models are not appropriate for programs with irregular parallelism, i.e. where important coordination characteristics like the number and granularity of tasks are determined dynamically. A systematic profiling-based methodology, however, can be used to develop effective irregularly-parallel programs for multiple architectures.

This paper presents and illustrates such a methodology for GpH, a semi-explicit parallel functional language. GpH is a language with both high-level computation, it is an extension of Haskell98, and high-level control, it automatically manages and distributes tasks, data and garbage, and to some extent dynamically adapts to the underlying architecture. Due to the complexity of implementing such a dynamic system, few robust effective multi-architecture implementations of languages with both high-level computation and control exist. We believe our multi-architecture development methodology to be the first for this class of languages.

We start by outlining the methodology together with the language and tools used (Section 2). The methodology is illustrated by the development of a substantial application (Section 3) for two very different architectures: a Beowulf cluster and a Sun SMP. It has the following phases: sequential implementation/optimisation (Section 4); idealised and realistic parallel simulation (Sections 5 and 6); tuning on the parallel architectures (Section 7). Finally, Section 8 concludes.

1.2 MULTI-ARCHITECTURE PROGRAMMING IN GPH

1.2.1 Parallel Programming in GpH

GpH [THLP98] provides parallel (`par`) and sequential (`seq`) composition as coordination primitives, and both simply return their second argument. Operationally `seq` causes the first argument to be evaluated before the second and `par` indicates that the first argument may be executed in parallel. The sophisticated language implementation dynamically manages many details of the parallel execution that are usually architecture-dependent, including task creation, communication etc.

Evaluation strategies [THLP98] abstract over `par` and `seq` to enable high-level specification of parallel coordination. A strategy is a function that may be higher-order, polymorphic or composed from other strategies.

1.2.2 GpH Compilers and Tools

Sequential Execution Tools: The *GHCi Interpreter* provides an interactive environment for fast program development, allowing the programmer to experiment and debug the sequential program. The *Glasgow Haskell Compiler (GHC)* [Pey96] is an optimising compiler for Haskell with numerous optimisation phases for reducing runtime and analyses that supply information about the program behaviour. The *Time and Space Profilers* [SP95] attribute execution time and memory allocation to expressions in the program, enabling the programmer to identify compute and memory-intensive components.

Parallel Execution Tools: The *GranSim Parallel Simulator* [HLP95] is a highly-parameterised simulator for GpH that allows the programmer to emulate different architectures, including an idealised parallel machine. It also provides a set of visualisation tools to help the programmer in understanding the dynamic nature of the execution. The *GUM Implementation of GpH* [THM+96] is a portable, parallel implementation of GpH, a parallel extension of the Haskell functional language. The implementation is message-based, and portability is facilitated by using the PVM communications harness available on many multi-processors. As a result, GUM is available for both shared-memory (e.g. Sun SPARCserver multi-processors) and distributed-memory (cluster) architectures.

1.2.3 GpH Multi-Architecture Programming Methodology

The new methodology is based on experiences developing a dozen substantial GpH programs, e.g. [LTH$^+$99, THLP98], and is summarised in Figure 1.1, where each node is a program, virtual machine pair. The methodology has two phases: an architecture-independent phase (upper half), that develops parallelism on a simulated idealised machine; and an architecture-dependent phase (lower half), that tunes the parallelism for a specific architecture.

FIGURE 1.1. A Multi-Architecture Program Development Model

The first stage develops an inherently-parallel program and optimises its sequential performance of the program using standard compilers and interpreters (Hugs, GHC), and their associated time and space profiling tools. This stage is particularly important in a high-level language like GpH where time and space costs are less apparent than in lower-level languages. In the second stage strategies are added to the program to produce an initial parallel program. The parallelism in the initial parallel program is refined using the GranSim simulator, parameterised to simulate an idealised machine. The idealised machine has, for example, zero communication costs, and an unbounded number of processors. The primary advantage of using an idealised machine is that we know that poor parallelism is inherent, and not an artifact of some specific architecture. If good parallelism cannot be achieved on the idealised machine it cannot be obtained on

```
Input sequences
   AUGCGAGUCUAUGGCUUCGGCCAUGGCGGACGGCUCAUU
   AUGCGAGUCUAUGGUUUCGGCCAUGGCGGACGGCUCAUU
   AUGCGAGUCUAUGGACUUCGGCCAUGGCGGACGGCUCAGU
   AUGCGAGUCAAGGGGCUCCCUUGGGGGCACCGGCGCACGGCUCAGU

Aligned output sequences
   AUGCGAGUCUA-----------UGG-CUU-----CGGCCAUGGCGGACGGCUCAUU--
   AUGCGAGUCUA-----------UGGACUU-----CGGCCAUGGCGGACGGCUCAUU--
   AUGCGAGUCUA-----------UGG-UUU-----CGGCCAUGGCGGACGGCUCA--GU
   AUGCGAGUC-AAGGGGCUCCCUUGG---GGGCACCGGC----GC--ACGGCUCA--GU
                         ^^^
                         |||
                    First Best Pin
```

FIGURE 1.2. Input Sequences and the Aligned Output Sequences.

any machine. The problem of obtaining good parallel behaviour is split into two parts: obtaining a good parallel algorithm, and adapting that algorithm to the performance characteristics of a specific architecture. We find in practice that most of the development work is done in the first phase and can be reused when targeting a new machine.

The architecture-dependent phase tunes the parallelism for a target architecture using the GranSim simulator, parameterised to emulate the target architecture. Key parameters include details such as the number of processors, communication latency, task creation overhead, and all are specified in terms of machine cycles, an abstract time measure. Typical changes during this stage are to adapt the parallelism to the characteristics of the target architecture, for example task granularity might need to be increased to offset creation overhead and message latency. The final stage is to measure and tune the program on the target architecture using the GUM implementation and profiling tools. Our experiences with the parallelisation of several programs indicate that this stage typically requires few changes [LTH^{+}99]. Normally the simulated results are a good approximation to the parallel behaviour under GUM. Typical changes during this stage are to adapt the I/O, or to utilise specific system calls on the target architecture. Of course, it is sometimes necessary to iterate through the development process. For example, the most efficient sequential version of an algorithm is not always the most efficient parallel version.

1.3 A GENETIC ALIGNMENT PROGRAM

The program studied aligns sequences of genetic material (RNA) from related organisms and has been described in [Bla00, FT89]. The alignment of a set of RNA sequences entails lining up the sequences with corresponding sections directly above one another. Special "indel" characters (-) are inserted to line up the sequences [San83] as shown in Figure 1.2.

Alignment algorithm. The input is a set of amino-acid $\{$A,C,G,U$\}$ `Sequences`. The alignment algorithm is based on the notion of *critical subsequences*: a subsequence of a single sequence that occurs only once within the sequence. When a critical subsequence occurs in two or more sequences, the set of occurrences is called a *pin*. To compute the best pin, all critical subsequences from each sequence are generated, and the their occurrences in the input sequences are counted. The critical subsequences with the highest number of occurrences is selected. *Unpinned sequences*, which are not aligned, arise if a sequence does not contain the best pin.

To align a set of sequences we compute the best pin and use it to partition all sequences into left, right and unpinned sequences. Then we recursively align the left, right and unpinned sequences. Finally we connect all pinned sequences with the best pin and append the unpinned sequences. Figure 1.2 shows an example.

1.4 SEQUENTIAL IMPLEMENTATION AND TUNING

The potential parallelism in a functional program is only constrained by the data dependencies between expressions. The first stage in the methodology develops a program that is sequential in the sense that it does not specify which of the many possible parallelisations to exploit.

Sequential implementation. The program has three major functions: `align_chunk`, `divide`, and `bestpin`. The `align_chunk` function is the top-level function and aligns a set of sequences (chunks) as described above. It calls `bestpin` to extract the best pin and `divide` to split the input into the left, right and unpinned chunks.
The `bestpin` function takes the input sequences and determines the best pin. Its logic comprises an *outer loop* function (`substring_sequences`) that generates all substrings from each sequence, and an *inner loop* function (`form_pin`) to compute the number of occurrences of each substring. The function `extract_max_pin` selects the pins which have the maximum number of occurrences in all sequences.

The `divide` function takes the input sequences and best pin, and uses the latter to split the input sequences into three chunks (left, right and unpinned) before recursively calling `align_chunk`. If no best pin can be found from any of the three chunks these chunks are merged.

Sequential Tuning. The sequential measurements show that storage management is a primary cost with garbage collection accounting for 120s of the 224s total execution time. The total allocated memory is 1034 MB with maximum residency is 12.25 MB. To improve the program a series of five optimisation are

TABLE 1.1. Summary of Sequential Tuning

	time			*memory*	
Prog.	Mut	GC	Total time	Max. Res.	Total Alloc
I	120.1s	104.0s	224.1s	12520Kb	141.8Mb
II	121.3s	100.8s	222.1s	12520Kb	141.8Mb
III	119.1s	100.8s	219.9s	12520Kb	141.7Mb
IV	37.3s	6.5s	43.8s	123Kb	1,349.0Mb
V	16.9s	1.9s	18.9s	123Kb	369.5Mb
VI	12.9s	72.9s	85.9s	13400Kb	128.0Mb

made, that are not all detailed here. Most of the changes aim to eliminate intermediate data structures. To eliminate the unpinned substrings at an earlier stage, the program computes the pin substrings in a sequence before it goes to generate the substrings from another sequences. This reduces the number of substrings that are carried to the next stage at the expense of introducing a tighter sequential dependency into the program. As a result of these modifications, the total memory allocated dropped to 707.55 Kb with maximum residency 122.70 Kb. The total execution time reduced to 18.92s comprising 17s of reduction (real execution time Mut) and 2s of garbage collection (GC). Table 1.1 summarises the results of sequential tuning, for all versions discussed in Section 5 and Figure 1.3 shows the space profile for the tuned sequential program.

FIGURE 1.3. Space Profile of Final Sequential Version.

1.5 IDEALISED PARALLELISATION

The next stage of the methodology is to develop parallel versions on an idealised machine. There are several sources of parallelism in the genetic alignment program and seven parallel versions are developed. Each version is measured on the GranSim simulator parameterised to emulate an idealised machine with zero communication costs and an infinite number of processors. The input data in each case is a set of 6 sequences containing 20 genomes. For the last three versions a "chunk size" of size 30 is used.

Version I: Divide and conquer parallelisation. Parallelism is introduced using a divide-and-conquer paradigm: the alignment of the left, right and unpinned chunks is independent so they can be computed in parallel. Only limited parallelism is generated, an average of 1.1 tasks. This is due to a large sequential component: the `bestpin` function on the initial input consumes about 78% of the runtime, which by Amdahl's Law limits the maximum speedup to 100%/78% = 1.28.

Version IIa: Parallelising substring sequences. This version parallelises the outer loop of the bestpin function described in Section 4. More specifically the `par_substring_sequences` function uses `parMap` to map the `substring_sequences` function over the input sequences in parallel.

Version IIb: Parallelising form pin. This version parallelises the inner loop described in Section 4. A new `par_form_pin` function is inserted and the `form_pin` function was modified to be executed in parallel over the supplied list.

Version IIc: Parallelise both substring sequences and form pin functions. This version combines inner and outer loop parallelism, i.e. parallelism from both Versions IIa and IIb.

Version III: Clustering in the parallel form pin function. This version includes all previous sources of parallelisation, (i.e. Versions I, IIa, IIb, IIc): also, data clustering was applied to the input list supplied to the `form_pin` function. The implementation of the `parMap` function on a collection of data such as a big lists often yields very fine task granularity. Clustering is one way to improve the task granularity and data locality by introducing fewer tasks, each operating on a closely-related subset of the collection.

Version IV: Parallel maps. This version extends Version III by replacing all map functions with `parMap`. In other words in this version we parallelise all intermediate functions called `bestpin` and `divide`.

TABLE 1.2. Idealised Simulation (Input: 20 6 30).

Prog.	Avg Par.	Spd up	Run-time Mcyc	Work Mcyc	Tasks	Avg Task Leng. Mcyc
Seq	1.0	1.0	139.6	139.6		
I	1.1	1.1	126.1	138.7	19	7.30
IIa	4.2	4.2	32.9	138.1	94	1.40
IIb	8.8	8.3	16.8	147.8	3542	0.04
IIc	13.6	9.5	14.7	199.9	3583	0.05
III	16.7	16.8	8.3	138.6	275	0.50
IV	21.1	21.5	6.5	137.1	381	0.35
V	**21.9**	**21.8**	**6.4**	140.1	785	0.17

Version V: Parallel folds. This version extends Version IV by adding a parallelised fold function in `extract_max_pin` function. We defined a new strategic function called `parfoldList` to execute the `foldr` function in parallel.

Results. The results obtained from the idealised stage are summarised in Table 1.2. The maximum speedup was obtained from Versions V (21.8) and IV (21.5), while other versions give more modest speedups. Versions IIa and IIb introduce a very large number of small tasks, with corresponding high task management overheads, reflected in the increase in total work.

FIGURE 1.4. Activity Profile of an Idealised Simulation of Version V.

Figure 1.4 shows the overall activity profile for version V on an simulated ide-

TABLE 1.3. Realistic 32-PE Beowulf Simulation (Input: 20 6 30).

Prog.	Avg. Par.	Spd up	Run time Mcyc	Work Mcyc	Tasks	Avg. Task Length Mcyc
Seq	1.0	1.0	139.6	139.6	–	–
I	1.1	1.1	127.2	139.9	19	7.40
IIa	1.9	1.7	80.8	153.5	94	1.47
IIb	1.2	0.2	708.9	850.7	3183	0.04
IIc	2.2	0.9	143.7	316.1	3201	0.04
III	2.0	0.9	150.8	301.6	275	0.50
IV	1.9	0.8	168.2	319.6	381	0.36
V	2.1	0.8	162.0	340.2	785	0.17

alised machine, with execution time on the X-axis and the number of tasks on the Y-axis. The tasks are separated into four classes, depending on their state: *running* if they are currently executing, *runnable* if they could be executed if a processor became available, *blocked* if they await data under evaluation, and *fetching* if they are retrieving data from another processor [Loi98, LTH$^+$99]. From the graph we see that for this input data the idealised machine could utilise approximately 40 PEs.

1.6 TUNING ON TWO SIMULATED ARCHITECTURES

The next stage of the methodology is to tune the program for a specific architecture, using realistic simulations. The following sections describe the tuning for both a shared-memory architecture and a distributed-memory cluster architecture. The architectures are simulated by parameterising GranSim with key architectural properties, most importantly the number of processors, the time to pack a message for transmission, and communication latency. Each property is measured in clock cycles of a target architecture processor.

Beowulf simulation. We use a 32-node 533MHz Pentium III Beowulf cluster connected by a fast ethernet switch. The PE-to-PE communication latency was measured as 142μs under PVM 3.4.2, so the Gransim latency is 142*533 = 75,300 clock cycles (75.3Kcyc). Likewise the packing time was measured as 21μs or 11Kcyc. Table 1.3 summarises the performance of the genetic alignment program versions on a simulated Beowulf, aligning 6 sequences of length 20 with chunk-size 30.

Sun SMP architecture. We use a 4-processor Sun SMP with clock speed of 250 MHz connected by a shared memory bus. The PE-to-PE latency under PVM is 109μs or 27.5Kcyc, and packing cost is 22μs or 5Kcyc. Table 1.4 summarises the

[h]

TABLE 1.4. Realistic 32-PE Sun SMP Simulation (Input: 20 6 30).

Prog.	Avg Par.	Spd up	Run time Mcyc	Work Mcyc	Tasks	Avg. Task Length Mcyc
seq	1.0	1.0	139.6	139.6	–	–
I	1.1	1.1	126.6	139.2	19	7.400
IIa	2.1	**1.9**	70.6	148.2	94	1.470
IIb	1.3	0.3	413.3	537.2	3542	0.042
IIc	2.9	1.4	97.6	282.0	3201	0.046
III	3.5	1.8	77.2	270.2	275	0.510
IV	3.3	1.7	81.4	268.6	381	0.360
V	3.4	1.7	81.6	277.4	785	0.170

FIGURE 1.5. Realistic simulation of Version IIa on Beowulf & SMP (Input: 20 6).

performance of the genetic alignment program versions on a simulated Sun SMP.

Idealised simulation vs realistic simulation. Comparing the idealised (Table 1.2) and realistic simulations (Tables 1.3 and 1.4) we make the following observations. For these small input sizes the speedup attained and utilisation of each architecture is extremely poor. The number of generated tasks is similar in all three simulations because most parallelism is flat data parallelism rather than the hierarchical parallelism produced by a divide-and-conquer paradigm. Both simulated machines give worse speedups than the idealised machines, with the simulated Beowulf being slightly worse than the simulated Sun SMP. This is caused by the increased communications latency, task management overheads and limited number of PEs in the realistic simulated machines. Important for the parallel program development, increasing the number of generated tasks always improves speedup in an

[ht]

TABLE 1.5. **Real 30-PE Beowulf (Input: 20 60 30).**

Prog	Speedup	Runtime (s)	Generated Tasks	Avg Task Len (ms)
seq	1	99.9	-	-
I	0.8	99.7	171	0.119
IIa	**7.5**	13.2	941	0.011
IIb	0.7	371.3	17011	0.003
IIc	1.0	94.0	36236	0.021
III	0.9	110.5	5096	0.057
IV	0.6	174.8	1289	0.013
V	1.0	116.2	1308	0.020

ideal machine, but this not the case on the simulated realistic machines because of the communication and tasks management overheads introduced.

Beowulf simulation vs Sun SMP simulation. Comparing the Beowulf and Sun SMP simulations in Tables 1.3 and 1.4 we make the following observations. Versions I and IIa have similar behaviour on both architectures as they generate a small number of large tasks compared with other versions. Figure 1.5 shows the activity profile of both architectures, Beowulf on the left and Sun SMP on the right. Both graphs are similar in shape except that the Beowulf shows more fetching tasks because of its higher latency.

In separate experiments we have obtained better speedups for both simulated architectures with larger input sizes, but time and disk space limitations of the simulation platform preclude larger systematic experiments. Comparing the speedups obtained from executing the different versions of the program on both architectures we find that even with a small input size the maximum speedup is 1.9 on the simulated Sun SMP and 1.73 on the simulated Beowulf, both for Version IIa.

1.7 TUNING ON TWO REAL ARCHITECTURES

The final stage of the methodology is to tune the program on the actual architecture, in our case a Beowulf and a Sun SMP. From Table 1.3, summarising the realistic simulation results on the Beowulf, it is clear that Version IIa delivers the best speedup. However, to validate the predictive ability of the realistic simulation, and to explore the differences between the simulated and real machines all versions are measured on both architectures. On both machines the input sizes are much bigger than for the simulations, therefore we cannot directly compare results from simulation and real measurement, i.e. Tables 1.3,1.4 with Tables 1.5,1.6.

TABLE 1.6. Real 4-PE Sun SMP & Beowulf (Input: 20 40 30).

Prog.	Sun SMP				Beowulf			
	Speedup	Runtime (s)	No. Tasks	Avg Task Len (ms)	Speedup	Runtime (s)	No. Tasks	Avg Task Len (ms)
seq	1	70.8	–	–	1	27.7		–
I	0.9	73.8	55	1.340	1.1	27.2	3	0.0250
IIa	**1.8**	37.9	616	0.061	**1.9**	15.0	21	0.003
IIb	0.2	332.0	23940	0.013	0.8	36.4	861	0.001
IIc	0.4	146.8	12036	0.012	1.5	18.6	16157	0.005
III	0.7	94.8	2585	0.036	1.3	20.9	601	0.007
IV	0.6	105.8	2850	0.037	1.5	18.4	601	0.004
V	0.8	87.6	3047	0.029	1.8	15.5	4226	0.001

FIGURE 1.6. Activity Profiles of Version IIa on a Beowulf & SMP (Input: 20 60).

Beowulf and Sun Measurements. The measurements of the programs on the 30 node Beowulf cluster described in Section 6 are given in Table 1.5, and configured with 4 PEs in Table 1.6. Table 1.6 also summarises the measurements of the programs on the 4-PE Sun SMP described in the same section.

Considering the different versions of the program reported in Tables 1.5 and 1.6, the best version is IIa on both architectures, with speedup 7.5 on the Beowulf and 1.8 on the Sun SMP. This is because Version IIa generates big tasks compared with Versions IIb to V. The worst version is IIb for both simulation measurements and real measurements, because of the large number of small tasks which increases the amount of communication in the program.

The maximum speedup obtained from input (20 60) was 7.5 when the program was executed on 30 processors of the Beowulf cluster. This speedup exceeds the speedups from the realistic simulation because we can run the program on larger inputs in the real execution, which increases the number of coarse grained tasks generated by this version, underlining the importance of task granularity.

FIGURE 1.7. Speedup vs Number of PEs (Beowulf & SMP real)

The difference in the number of generated tasks is a result of GUM's dynamic task distribution mechanism. The Beowulf has higher communications latency than the Sun SMP, so a PE needs more time to obtain work from other PEs. This difference shows how the GUM implementation of GpH dynamically adjusts the granularity of the parallelism to the specific parallel architecture.

Table 1.6 shows that both 4-PE architectures deliver similar speedups, i.e. 1.8 on the Sun SMP and 1.9 on the Beowulf for version IIa. Figure 1.7 shows the speedup graphs obtained from Beowulf and SMP.

1.8 SUMMARY AND CRITIQUE OF THE METHODOLOGY

A profiling-based methodology for developing irregularly-parallel applications for multiple architectures in a semi-explicit parallel functional language has been presented. The methodology is the first for languages with both high-level computation and coordination, and has been illustrated using a genetic alignment program. The key points learnt from investigating the methodology are as follows.

Sequential profiling is crucial for good overall performance and independent of parallelisation. Sequential execution time dropped from 224s to 18.91s. The idealised parallel version can be reused when targeting new architectures, reducing programming effort. The GranSim simulator provides considerable flexibility to emulate different architectures including the idealised machine.

The idealised simulation results (Table 1.2) show that improved speedup corresponds to increased task generation. However many of the new tasks are small, and while they may improve performance on a low-latency architecture, they degrade performance on a high-latency architecture (Table 1.5). While GUM can dynamically adjust the number of tasks (Table 1.6 exhibits fewer tasks on the high-latency Beowulf), its mechanism is more effective in divide-and-conquer programs compared to programs with a flat parallel structure. Thus with a high-latency machine and flat parallelism, versions with fine grain tasks (i.e. IIc and onwards) should be avoided.

Realistic GranSim simulation correctly predicts the program versions that will deliver a good speedup on different architectures. However, there are differences between the activity profiles produced from GranSim and GUM, as shown in Figures 1.5 and 1.6. While the simulator is a good tool for guiding the parallel program development process, it is too simplified to provide performance prediction data that could be used for automatic parallelisation of sequential programs.

While the performance of the alignment program developed here is only just acceptable (7.5 on a 30-processor Beowulf and 1.8 on a 4-processor Sun SMP), far better speedups have been achieved for other programs using the emerging methodology: e.g. 12.0 on a 16-processor workstation cluster and 2.8 on a 4-processor Sun SMP for a data-intensive accident analysis program; 11.9 on a 16-processor Beowulf for a linear equation solver [LTH+99]. Hence, we attribute the modest raw performance for the genetic alignment program to limitations in the parallelism of the application, rather than to shortcomings of our methodology.

In assessing our methodology we conclude that most of the parallel program development can be done in an architecture independent fashion, using familiar sequential tools and simulators for parallel execution. Due to GpH's high-level coordination and GUM's automatic adaption to the underlying architecture, no changes to the genetic alignment program are required to move it from a Beowulf cluster to a Sun SMP architecture. Moreover the best performance on both architectures is obtained from the same parallel version of the program. The overall performance achieved is acceptable, considering that the parallelisation of the program required only minimal code changes in the first place. Using the methodology has also suggested concrete improvements, namely that on high-latency architectures program versions with flat parallelism and fine task granularity are not developed in the idealised parallelisation phase.

1.9 REFERENCES

REFERENCES

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

[BJK+95] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP'95 — Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 19–21, 1995.

[Bla00] J. Blazewicz. *Handbook of Parallel and Distributed Processing*. Springer, 2000.

[Ble93] G. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[CC00] L. Chamberlain and E. Christopher. ZPL: A Machine Independent Programming Language. *IEEE Transactions on Software Engineering*, 26:197–212, March 2000.

[FT89] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1989.

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — Conference on High Performance Functional Computing*, pages 208–221, Denver, CO, April 1995.

[KLSS94] C. Koelbel, D. Loveman, R. Schreiber, and JR. Steele. *The High Performance Fortran Handbook*. The MIT Press, March 1994.

[KRSG94] L.V. Kale, B. Ramkumar, A.B. Sinha, and A. Gursoy. The Charm Parallel Programming Language and System. *IEEE Trans. on Par. and Distr. Sys.*, 1994.

[LFA96] D.K. Lowenthal, V.W. Freeh, and G.R. Andrews. Using Fine-Grain Threads and Run-Time Decision-Making in Parallel Computing. 37(1):41–54, 1996.

[Loi98] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, March 1998.

[LTH$^+$99] H-W. Loidl, P.W. Trinder, K. Hammond, S.B. Junaidu, R.G. Morgan, and S.L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):701–752, October 1999.

[MSBK01] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 16:181–206, 2001. Special Issue on High Level Models and Languages for Parallel Processing.

[Nik94] R.S. Nikhil. Cid: A Parallel "Shared-memory" C for Distributed Memory Machines. In *Workshop on Languages and Compilers for Parallel Computing*, volume 892, pages 376–390, Ithaca, NY, August 1994.

[Pey96] S.L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP'96 — European Symp. on Programming*, pages 18–44, 1996.

[San83] D. Sankoff. *Time Warps, Spring Edits and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[SP95] P.M. Sansom and S.L. Peyton Jones. Time and Space Profiling for Non-strict Higher Order Functional Languages. In *POPL 95 — Symp. on Principles of Programming Languages*, 1995.

[ST98] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[THM$^+$96] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Language Design and Implementation*, Philadephia, USA, May 1996.