NTNU
Innovation and Creativity

Dynamic Selection of MPI Intra-copy Routines Based on Program Characteristics

Øystein Lauen Borg

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Håkon Bugge, Scali

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

When, for instance, calling MPI_Send the underlying system may copy the data to either the calling processes' local memory, the receiver's memory, or both. The performance of this copy depend not only on the parameters of the MPI call, but also the state of the calling program and the underlying system. Consequently, it is necessary to taylor library routines such as the MPI_Send fuction to the given calling program.

In this thesis, we will evaluate how to select copy functions in order top optimize some MPI calls. The methods developed will be tested on some NAS kernels, or other suitable benchmark.

Assignment given: 20. January 2006
Supervisor: Anne Cathrine Elster, IDI

Preface

This report is a Master's Thesis at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

It was written by Øystein Lauen Borg during the spring 2006. The thesis problem description was given in collaboration with the company Scali AS, who let me use and modify the source code of their product Scali MPI Connect.

Thanks to my supervisors, Dr. Anne Cathrine Elster and Håkon Bugge, for giving me helpful advice as well as making this thesis possible. I would also like to thank Åsmund Østvold who work at Scali AS for valuable ideas, advice and support during the course of my work.

Trondheim, June 2006

Øystein Lauen Borg

Abstract

The *Message Passing Interface* (MPI) has become a de-facto standard for parallel programming. The ultimate goal of parallel processing is high performance and this brings a motivation for a highly optimized MPI - implementation.

When an application calls an MPI communications routine, data is copied between user memory and the memory areas managed by the MPI library. The speed of this transfer depends on a multitude of factors, including the architecture, amount of data, data layout and whether the data is referenced right before or after a transfer. There are numerous ways to copy data from one location to another, and their characteristics combined with the data properties will yield different efficiency. The information needed to select the best way to copy data is only available during application execution.

In this Master's Thesis, we present and implement a method to improve the performance of parallel applications by dynamically perform a close-to-optimal selection of intra-copy routines within an MPI implementation. Our method detect loops of MPI calls, and exploit loop predictability to time their performance while varying the routine selections. In order to obtain a good routine selection reasonably fast, a global optimization heuristic, simulated annealing, is used.

In particular, our solution method is employed within Scali MPI Connect (SMC), an MPI implementation providing 35 different intra-copy routines. Through various benchmarks, it is observed that our method introduce low overhead and find a good selection fast, thus reducing the execution time of the given benchmark. In benchmarks where the difference between an optimal routine selection and the standard selection within SMC allows it, a bandwidth improvement of 40% is observed.

Keywords: Parallel computing, memory copy, loop detection, simulated annealing.

Contents

List of Figures

Chapter 1

Introduction

In the last decade, message passing has become the predominant programing model for parallel applications with the *Message Passing Interface* (MPI) as the *de facto* standard. The majority of MPI programs are scientific applications that are parallelized for performance. For instance, one might want to solve increasingly complex problems without exceeding a given time slot. This motivates the utilization of several processors for problem solving in order to obtain satisfactory wall-time performance.

A typical real-world example is numerical weather predication, where the accuracy of the prediction depends upon how recent the input is as well as the resolution of the model area, while its usefulness relates to the ratio between time spent on simulation and how long into the future the prediction goes. Weather prediction applications hence requires the completion of the calculations to happen within a given time slot with a problem size as large as possible. Given today's technology it is thereby essential to parallelize this applications in order to get the performance needed.

Since the applications are built on MPI, it is vital that the MPI-library is optimized to obtain a performance gain when employing multiple processors. Today, there exists numerous MPI implementation and the majority of them are highly optimized. However, these optimizations are based on clever programming and general estimates about the typical MPI application and the underlying hardware. Many algorithms internal to the MPI implementation are statically assigned during compilation, and the selection of which ones to use is based on the architecture and operating system used at compile time.

The MPI implementations Scali MPI Connect [Sca], LAM/MPI [BDV94] [SL03] and MPICH2 [GLDS96] all have a vast number of internal algorithms to choose from. These algorithms are set to predefined default values at application start-up. However, the user can control the algorithm selection by exporting environment variables.

To set these variables properly, the user need knowledge of the characteristics of the application, the MPI implementation and the hardware used. The algorithm selection affected by the setting of environment variables persist until program termination, regardless of any changes or variations in the communication pattern.

It is clear that there is a room for improvement of these MPI implementations by dynamically selecting internal algorithms based on run-time performance metrics. Optimal selection will most likely further

reduce the overhead introduced by communication.

1.1 Problem Statement - Run-time Optimization of MPI

In this thesis, we want to do run-time optimizations of an MPI implementation. In particular, we want to optimize the selection of intra-copy routines within Scali MPI Connect (SMC) and assess the impact of this on MPI benchmarks.

Intra-copy routines are used to copy data between user-memory and memory locations managed by the MPI-library. The speed of this transfer depends on a multitude of factors, such as architecture, amount of data, data striding and when the data is used in the application. There are numerous ways to copy data from one location to another, and their characteristics combined with the data properties will yield different efficiency. In particular, there are 35 different intra-copy routines provided within SMC.

There is a correlation between where in the memory hierarchy data is located and the speed of a copy routine, consequently there is a correlation of the efficiency between subsequent copy-routines. This lead to even more complex optimization.

1.2 Our Solution Method

The information we need to do the selections are only available during application execution. Hence, we need to empirically measure the performance of the copy routines with a given application and underlying hardware. Our solution is to exploit the fact that the most common control structure in a program is a loop. Loops have properties that are ideal for measurements. For instance, the work performed within a loop are likely to stay constant, and several loop iterations lead to credible measurement results. By tracking the return address of each MPI call, we are able to detect loops in the application program, either by dynamically detecting simple reducible loops, as suggested in [TG98] or by constructing a control flow graph during a "dry run" and then retrieve loops by the methods described in [CHK01] and [SGL96]. These loops are then separately and exclusively timed while varying the memory copy routines used. In order to find a good solution fast, a global optimization heuristic, like simulated annealing described in [PTVF92] and [Ing89] is used. Based on prior empiric results and theoretical properties the solution space for the heuristic can be substantially reduced in advance.

The novelty of our solution method is the detection and utilization of loops of library calls to do run-time optimization. The solution space for the simulated annealing will dynamically evolve since the timing result of a solution will vary due to random fluctuations. The use of simulated annealing on such a solution space is also new to our solution method.

1.3 Thesis Outline

The overall plan is to implement and apply our solution method to parallel benchmark programs in order to evaluate its performance.

Chapter 2 give a brief description on related work, while chapter 3 present the background theory needed to implement our method. Relevant topics like loop detection and simulated annealing are covered, as well as background material on MPI and statistical methods for performance measurement.

Chapter 4 give details on the implementation of the solution method. The new algorithm for run-time detection of loops of MPI calls, and use of simulated annealing on an evolving solution space are presented.

In Chapter 5 the benchmarks utilized to evaluate properties of our solution method are described. Known sources of measurement deviation are also included.

Chapter 6 show and discuss properties of our solution method based on results obtained by running the benchmarks described in Chapter 5.

Finally, Chapter 7 conclude the work done in this Master's Thesis and point out topics for further investigation.

Appendix A gives a brief summary of the statistical method ANOVA, which is used to gain confidence in our measurements, while Appendix B gives a short description of the code that was implemented.

Chapter 2

Related Work

Much work has been done in order to optimize the internal workings of MPI implementations.

[FY05] have chosen an approach inspired by Fastest Fourier Transform in the West (FFTW) [FJ05] and Automatically Tuned Linear Algebra Software (ATLAS) [WD98], where the idea is to try out a multitude of small routines and variations of them to find the set of best suited routines and parameters for the specific architecture.

[OM96] and [KYL03] are examples of MPI optimization by using information known at compile time. The first use this information to remove all unnecessary MPI overhead and create an MPI which is specialized to the particular application. The latter apply aggressive optimizations on communication calls whose parameters are known.

Dynamic improvement on MPI has been done in [HZKK06]. They use virtual MPI processes (VPs), and exploit the added freedom to adaptively assign various VPs to different processors at run-time in order to reduce latency, overlap calculation and communication, and achieve dynamic load balancing. [Nat05] optimize MPI point to point communication by replacing synchronous calls with asynchronous. This is done at run-time by dynamic detection of requests to the involved message buffers. Finally, [NPSC] dynamically optimize selection of collective operations at run-time through a lottery scheduler.

Loop recognition and detection in graphs is a well known field of research and is utilized in compilers to unroll loops of basic blocks and apply optimization techniques like loop level parallelism analysis. [TG98] and [dAK01] do run-time detection of loops at instruction level. Both exploit the predictability of loops, [TG98] to speculate future instructions and [dAK01] to characterize loop execution in order to present a guide in the selection of design parameters of a loop path predictor.

Simulated annealing (SA) has been used to solve a wide range of combinatorial optimization problems, from the Traveling Sales Person (TSP) [PTVF92] to filtering of binary images [GKR94]. However, no literature has been found that describe use of SA in a context where the solution space has dynamic behavior and with hard execution time constraints. In fact, the literature is dominated by asymptotic convergence properties rather than finite time practical use of SA [HJJ03].

Chapter 3

Background Theory

This chapter presents the background material and the theory utilized in later chapters. First, a brief overview of MPI is given along with some main aspects of its communication modes. Then, some theory on memory copy operations followed by a study of loop detection and simulated annealing. Finally, a description of the statistical knowledge needed in order to measure computer performance is given.

3.1 MPI

The Message Passing Interface (MPI) [For94] is a specification of a library of functions that can be called from a C, C++ or Fortran program. The essence of this library is a group of functions that can be used to achieve parallelism by explicit message passing. These functions simply transmit data from one process to another, or do collective data transfer and manipulation within a group of processes.

3.1.1 Communication modes

MPI provide a send and a receive primitive for point-to-point communication. These are extended to support various communication modes. Also, they are combined in different fashions to operate on groups of processes.

Point-to-point Communication

Basic message passing is implemented in the two routines MPI_Send and MPI_Recv. Both require "envelope" information that identify the corresponding process along with a pointer to the send/receive buffer and a description of the data layout.

There are several versions MPI_Send, the normal operation copy user data to a pre-allocated buffer maintained by the MPI implementation before it continue execution regardless of the actual data transfer. Another version, MPI_Ssend, does not return control until a matching receive has been posted and data reception has begun. MPI_Rsend require that a matching receive has been posted before it is called. Finally, we have MPI_Bsend which use user-allocated buffer.

These are examples of *blocking* operations, which means that the arguments of the function call can safely be modified by subsequent instructions. There are also *non-blocking* versions of the send operations and a non-blocking version of MPI_Recv, MPI_IRecv.

Collective Communication

Send and receive primitives can be combined such that a collection of processes can do collective operations. These operations scatter and/or gather data, and may even do calculations on the communicated data. $MPI\_Allgather$, $MPI\_Alltoall$, MPI_Gather, $MPI\_Scatter$, MPI_Bcast and to some extent $MPI\_Barrier$ only do data transfer, while $MPI\_Allreduce$, MPI_Reduce, $MPI\_Reduce\_Scatter$ and MPI_Scan allows for operations on their operands.

The basic requirement is that all processes in a specific collection, called communicator, participate in the collective operation. If any process is left out, the function call will most likely never return.

Derived Datatypes

Not all user-data to be sent are layed out continuous in memory. Instead of having the user go through the troublesome task of structuring the data before and after a transfer, MPI provide built-in datatypes and functionality to construct user-defined datatypes. Any kind of structure is allowed, and even recursive datatypes can be created. The MPI implementation handle the data layout induced by the datatype at communication calls, often involving numerous memory copy operations.

3.2 Memory Copy

Memory copy is a basic function which is frequently used in many applications. Often, a large quanta of an applications execution time is spent in memory copying routines. Therefore, memory copy is a common subject for optimization.

Many applications use the generic $memcpy$ routine which is a part of the standard ANSI C library. However, the efficiency of a memory copy depends a good deal on the architecture, and the generic memcpy does not necessarily utilize the processor and cache to their full extent.

This section will give an overview of some of the issues to address when implementing a fast memcpy routine. The topics are specifically aimed at IA32[1], IA64[2] and x86-64[3] architectures, but may be generally applicable. Background material is found in the manuals [Int04] and [AMD04].

[1] IA32 is Intel Architecture, 32-bit and is the instruction set of many of Intel's microprocessors. Generically called x86-32, but also referred to as i386.

[2] IA64 is Intel Architecture, 64-bits. Another instruction set from Intel, for 64-bit processors.

[3] X86-64 is a 64-bit microprocessor architecture. AMD64 is a 64-bit microprocessor architecture and a corresponding instruction set developed by Advanced Micro Devices (AMD).

3.2.1 Memory Alignment

In order to fully utilize cache, data memory should be aligned to an address which is dividable with a power of 2. This ensure that cache lines will be full of useful data. Non-aligned data are likely to result in superfluous cache loads.

Memory alignment also ensure that block-copying routines can execute without any additional overhead of post and preblock operations.

3.2.2 Extended Instruction Sets

We have a series of instruction sets that may be used to enhance the performance of memory copy. Along with the extended instruction sets there are additional registers to use them on.

MMX A SIMD[4] instruction set from Intel. Add 8 new registers to the architecture addressed from MM0 - MM7. Use the concept of packed data types allowing quad words to be stored in each of the 64-bit registers.

SSE Streaming SIMD Extensions, an instruction set from intel as a response to 3DNow! from AMD. Now supported by both. SSE introduce 70 new instructions and eight new 128-bit registers known as XMM0 - XMM7.

SSE2 Another ia32 instruction set introduced with the Pentium 4. It has additional operations on the XMM registers and includes a set of cache control instructions intended to minimize cache pollution. AMD's implementation on AMD64 includes eight more XMM registers for a total of 16.

The additional registers and instructions does improve the memory copy operation by allowing more data to be read at a time, see [Int04] and [AMD04] for an overview of the mov instructions. There are also more efficient ways to write data to memory through *write-combining*, which allow multiple writes to adjacent locations to be assembled and written into the cache hierarchy as a unit, saving both port and bus traffic.

3.2.3 Prefetching

The processor is able to prefetch cache lines before it actually needs them for the load, thus reducing the waiting time for the data and avoiding load penalty. Prefetching can provide significant gains, especially for regularly strided data access.

There are both hardware and software prefetchers. The hardware prefetcher automatically detect strided accesses and prefetch data as necessary. There is an associated start up and termination cost with the hardware prefetcher since it takes a few accesses to start it, and it will fetch too many cache lines at the end. These problems does not exist with the software prefetcher, but the programmer has to carefully select how far ahead the prefetch should be. Too late or too early prefetch may degrade performance.

[4] Single Instruction, Multiple Data

There are several software prefetch instructions, allowing the programmer to decide which cache levels to use. The Intel manual [Int04] classifies the data reference patterns as follows:

Temporal : data is going to bee used again soon.
Spatial : data will be used in adjacent locations.
Non-temporal: data which is referenced will not be used again in the immediate future.

For each of these reference patterns, there is a prefetch command. For example, there is the $prefetch\_nta$ command for non-temporal data which will not pollute cache with the data referenced.

This has an impact on the efficiency correlation between memory copy routines and data access patterns in an application. If data is copied but not referenced again in a long time, a typical scenario for sending data, the non-temporal instruction should be used. If the data will be used immediately after the copy, a scenario for receiving data, it is beneficial that the data resides in cache.

3.2.4 TLB Priming

The Transaction Lookaside Buffer (TLB) is a fast memory buffer that is used to improve the performance of translation between physical and virtual memory addresses. It typically use a virtual address as a search key and the search results in a real or physical address. If no match is found, we have a *TLB miss*.

When a TLB miss occurs, the translation is continued to the page table which has to be read from memory. The new entry is then stored in the TLB. A miss gives a performance degradation because of the additional memory access. To avoid TLB misses, the TLB can be preloaded with the next desired page table entry by touching an address in that page. This is called *TLB priming*. It ensure that the next page table entry is being loaded in advance of its use, and that a prefetch instruction will work correctly. If we have a TLB miss, a prefetch instruction might not have any performance impact at all.

3.2.5 Copy Routines

The topics discussed here are used in combination to obtain a fast memory copy operation. The internal memory copy routines found in Scali MPI Connect make use of the following principles:

- Use of assembly instructions to gain full control of registers. In particular the MMX and XMM registers combined with their extended instruction sets MMX, SSE and SSE2.
- Use of streaming instructions to write directly back to memory without touching cache and also getting out of order execution.
- TLB priming is used ahead of prefetching to ensure that the page entry is present in the TLB.
- Prefetch instructions is used to get data before it is actually needed.
- Copy data backwards to avoid the hardware prefetcher and possible associated prefetch penalties.
- Memory is aligned for use of block copying instructions and optimal use of cache lines.
- Loading and storing data are ordered in particular ways. Either by alternate between load and store instructions or by grouping the loads and stores separately, allowing for write-combining.

3.3 Loop Detection

Loops are a very common control structure in every program. A high percentage of all instructions executed in a program belong to loops [TG98]. Since loop branches and the amount of calculations done within a loop is highly predictable, they are potentially useful to perform run-time modifications and immediately measure the impact.

In order to exploit these nice properties we need to detect loops. In Chapter 2 we learn that loop detection is used for basic blocks in compilers, and at instruction level for speculation. In this case, we detect loops based on the sequence of MPI communication calls from within an application. The terminology, however, is the same. A loop is roughly described in [TG98] as all the instructions in a loop body, with one or more backward branches $B_1, B_2, .., B_n$ to a common instruction called the target T. All iterations of the loop after the first backward branch starts at T and ends at one of the branches. Such loops can be represented as a flowgraph where every instruction is a node and every edge is the jump from one instruction to another.

This translate the problem of detecting loops in program code to loop detection in graphs. One popular paper on this subject is Identifying Loops Using DJ Graphs [SGL96], which is based on the classic technique for detecting loops, namely Tarjan's interval-finding algorithm [Tar73].

In the following sections, the means to identify loops as described in [SGL96] and [CHK01] are presented.

3.3.1 Notation and Definitions

The following is a short introduction of the notation and the definitions used in loop detection.

- A flowgraph is a directed graph $G = (N, E, START, END)$, where N represent all nodes in the graph and E all the edges. $START$ is an unique node with no incoming edges and END an unique node with no outgoing edges. All other nodes has incoming and outgoing edges, and $START, END \in N$.
- If $x \rightarrow y \in E$, then x is the source node and y the destination.
- A path of length n is a sequence of edges $(x_0 \rightarrow x_1 \rightarrow ... \rightarrow x_{n-1})$ where each $x_i \rightarrow x_{i+1} \in E$.
- A node x dominate y if all paths from $START$ to y pass through x. Denoted by x dom y. By definition x dom x.
- A node x strictly dominate y if x dom y and $x \neq y$. Denoted by x stdom y.
- A node x immediately dominate y if x stdom y and there is no x' so that x stdom x' stdom y. Denoted by x idom y.
- A dominator tree is a tree where every edge represent a relation x idom y.

This give the means to build a dominator tree from a flowgraph. Further, we can build a DJ graph from the dominator tree and the flowgraph.

The DJ graph of a flowgraph consist of the same set of nodes as the flowgraph, and a super set of the edges. The edges are classified into two disjoint sets called D edges and J edges. D edges are the dominator tree edges. All other edges from the flowgraph is a J edge.

We distinguish between two types of J edges: back J edges (BJ) and cross J edges (CJ). An edge $x \rightarrow y$ is a BJ edge if y dom x, otherwise it is a CJ edge.

Figure 3.1(a) illustrate a simple flowgraph with two nested loops. Figure 3.2 show the DJ graph derived from that flowgraph. The dashed lines are dominator edges from the dominator tree. The other two edges are BJ edges as both of the source nodes are dominated by their respective destination node.

By performing a depth-first-traversal we can derive a depth-first spanning tree of the DJ graph. This traversal will classify each edge $x \rightarrow y$ as one of four types:

sp-back : if $y = x$ or y is an ancestor of x in the spanning tree.
sp-cross : if x and y have no ancestor-descendant relationship.
sp-forward : if x is an ancestor but not the parent of y in the spanning tree.
sp-tree : if x is the parent of y in the spanning tree.

A dept-first-traversal on Figure 3.1(a) will resolve that all dominator edges are sp-tree edges and the BJ edges are sp-back edges. The numbering on the right side of the DJ graph show the depth of each node in the dominator tree.

3.3.2 Loop Definition

Given a flowgraph or a DJ graph, we need the definition of a loop in a graph.

> *A loop is a strongly connected subgraph of a flowgraph. There are two kinds of loops; reducible and irreducible.*

The definition of reducible and irreducible loops is given in Characterizations of Reducible Flowgraphs [HU74]:

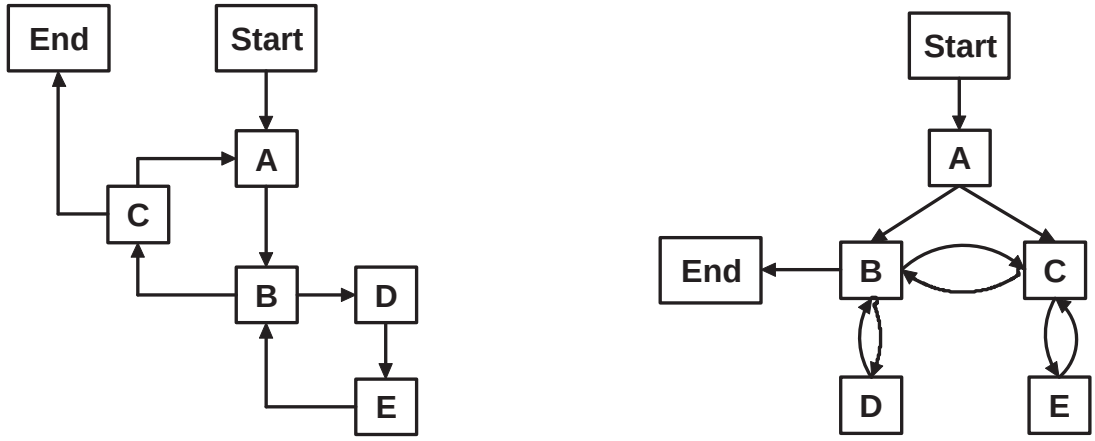
A graph G is reducible if and only if we can partition the edges into two disjoint groups, called the forward edges and back edges, with the following two properties:

1. The forward edges form an acyclic graph G in which every node can be reached from the start node of G.

2. The back edges consist only of edges whose destination nodes dominate their source nodes.

All other loops are irreducible. One of the key features of reducible loops, is that they do not have any incoming edges from nodes outside the loop, and the entry node dominate all other nodes in the loop. Irreducible loops have multiple entry nodes where none of them dominate all the nodes in that loop, however, they collectively dominate the loop.

The figures 3.1(a) and 3.1(b) display flowgraphs which contain reducible and irreducible loops respectively. Figure 3.1(a) is clearly a reducible flowgraph. The dominator edges in Figure 3.2 is the set of forward edges, while the final two are backward edges.



(a) Reducible flowgraph (b) Irreducible flowgraph

Figure 3.1: Examples of reducible and irreducible flowgraphs

Figure 3.1(b) show an irreducible flowgraph. None of the edges between B and C has a source node that is dominated by its destination and neither of them can be included in the acyclic graph formed by the forward edges. Thus, they violate the second property needed to classify the graph as reducible.

3.3.3 Loop Detection Algorithm

In this section, the algorithm suggested in [SGL96] is presented.

Given a reducible flowgraph, all sp-back edges are BJ edges, and all other edges are sp-forward or sp-tree edges. Hence, no matter how the depth-first traversal on the DJ graph is performed, the same set of sp-back edges will be found. This property is exploited in this algorithm to find reducible loops in an elegant procedure.

Irreducible flowgraphs invalidates the nice properties of reducible graphs. Different dept-first-traversals can resolve in various sets of sp-back edges. In Figure 3.1(b) one of the two edges $B \rightarrow C$ and $C \rightarrow B$ will become an sp-tree edge while the other become an sp-back edge, depending on the traversal order.

The algorithm utilize the following graph-theoretic result in order to detect irreducible loops:

> A flowgraph is irreducible if and only if there exists a simple cycle in its DJ graph that does not contain a BJ edge.

That is, the cycle is made of only D and CJ edges. Hence, one of the edges has to be both a CJ edge and an sp-back edge. It is further proven in [SGL96] that all the nodes in an irreducible loop can be found by performing the Strongly Connected Component algorithm on the nodes with a certain dominator tree level.

The dominator tree level is used to detect the ordering of loops, and to find reducible loops contained inside irreducible loops. Inner loops will always have an entry node whose dominator tree level is higher than the entry node of an outer loop. Algorithm 1 explore the DJ graph from the highest dominator tree level, and "collapse" loops into the entry node when found. Applied at the flowgraph from Figure 3.1(a), the algorithm would perform the follwing steps:

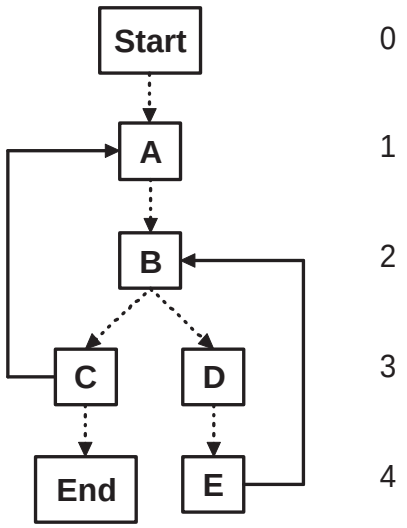


Figure 3.2: The DJ graph of the flowgraph in Figure 3.1(a)

1. First notable event when $n = B$ at line 5. The edge $E \rightarrow B$ is found to be a BJ edge at line 10.
2. At line 11, D and E is found without going through B and the loop consisting of B, D, E is collapsed into B.
3. The edge $C \rightarrow A$ is detected as a BJ edge, and the loop consisting of A, B, C is collapsed into A.

Both loops are found, and their nesting order are now known. Equally, on the flowgraph from Figure 3.1(b), the two reducible loops B, D and C, E are found while the irreducible loop is ignored and collapsed.

Algorithm 1 is used in this thesis in order to detect loops. It is suitable since it is able to record loop ordering and detect all reducible loops. The Strongly Connected Components algorithm detect all subgraphs in a directed graph in which every node can reach all other nodes. Algorithm 2 [5] is proven correctly in [CSRL01].

3.3.4 Dynamic Detection

The algorithm covered in the last section require that a full flowgraph has been produced. This can be obtained by a "dry run" of the application, but it require that the application has deterministic control flow and is not affected by variations in the input or background load. This is actually quite likely for many parallel applications, and particularly for those described in [BBB+91]. However, [HZKK06] claims that the new generation of parallel applications are complex, involve dynamically varying systems, and use adaptive techniques to fully exploit the processing capacity. This implicate non-deterministic control flow and bring the motivation to detect loops dynamically, since they may

[5] It use the notation G^T as the *transpose* of a directed graph $G = (N, E)$. That is the graph $G^T = (N, E^T)$ where each edge $x \rightarrow y$ in E is reversed to $x \leftarrow y$ in E^T.

Algorithm 1 Identify loops in a flowgraph as described in [SGL96].

1: Construct a DJ graph from the flowgraph.
2: Perform a depth-first traversal on the DJ graph to identify sp-back edges.
3: **for** $i = NumberOfLevels - 1$ downto 0 **do**
4: $IrreducibleLoop \leftarrow false$
5: **for all** nodes n at level i **do**
6: **for all** incoming edges $m \rightarrow n$ **do**
7: **if** $m \rightarrow n$ is both a CJ and an sp-back edge **then**
8: $IrreducibleLoop \leftarrow true$
9: **end if**
10: **if** $m \rightarrow n$ is a BJ edge **then**
11: Find the set S of all nodes that can reach m without going through n.
12: Collapse the loop consisting of $S \cup n$ to a single node.
13: **end if**
14: **end for**
15: **end for**
16: **if** $IrreducibleLoop$ **then**
17: Identify SCCs for the sub graphs induced by nodes at level $\geq i$.
18: Collapse each nontrivial SCC to a single node.
19: **end if**
20: **end for**

Algorithm 2 Strongly-Connected-Components

1: Compute a postfix numbering of a dept-first-traversal of the flowgraph G.
2: Compute G^T.
3: In decreasing postfix number, do a depth-first-traversal from each node.
4: Output the nodes of each three in the depth-first forest formed in line 3 as a separate strongly connected component.

differ between executions. Also, dynamic detection can be done without any intervention, like a "dry run", from a user.

Dynamic loop detection is described in [TG98] as a way to exploit the highly predictable behavior of loops and their history to speculate the future instruction sequence.

The main idea is to record the instructions $B_0, ..., B_n$ that jumps to a target T, and classify all the instructions in between as member of the particular loop started at instruction T.

A loop has the following run-time boundary conditions:

Startup The loop is initiated when an instruction whose address belong to the loop body is executed.

Termination The end of a loop is determined by one of the following conditions.

1. A not taken branch B_j
2. A taken branch or jump from inside the loop body to outside the loop body.
3. A return instruction whose address belong in the loop body.

This give the following characteristics of a loop execution:

- The first iteration of a loop is initiated by the start-up property above.
- All subsequent iterations of the loop start at address T.
- All iterations of the loop except the last one ends at one of the backward branches $B_0, ..., B_j$.

The immediate consequence of this is that nested loops and recursive calls will be correctly identified. However, the obvious weakness is that a loop is not detected until it has been executed at least once. Instructions may be shared between loops, resulting in overlapping loops which is an unwanted property when timing their performance.

3.4 Global Optimization

Global optimization is the minimization[6] of a function f that may depend on one or several variables. Or more formally, let Ω be the *solution space*, and $f : \Omega \rightarrow \Re$ the *objective function* defined on the solution space. The goal is to find a global minimum, ω' such that $f(\omega) \geq f(\omega')$ for all $\omega \in \Omega$.

The problem of finding a global minimum is considered very hard [PTVF92] since the solution space may grow exponentially with the number of variables. For a sufficiently large problem, an exhaustive search algorithm will be useless, so for all practical applications, a search heuristic or meta-heuristic is used.

The key feature of a global optimization heuristic is the ability to escape local optima by hill-climbing moves which worsen the objective function value, but give means of finding a global optimum.

Randomized search heuristics like randomized local search, the Metropolis algorithm, simulated annealing, evolutionary algorithms, tabu search, and genetic algorithms are all able to find an optimum and they apply to a large group of problems because of their generality.

[6] Or maximization, the solution method should be independent of the objective function.

In this thesis, the simulated annealing meta-heuristic is used because of its ability to converge fast and make use of advanced probability distributions while it retain simplicity in implementation.

3.4.1 Simulated Annealing

Simulated annealing (SA) is a generalization of a Monte Carlo method suitable for discrete optimization problems of large scale, where a desired global extreme is among many, poorer, local extrema [PTVF92]. The heart of the method is the analogy with thermodynamics, namely in the way metals cool and anneal. At high temperatures, the molecules of metals move freely with respect to one another. While the metal is slowly cooled, the thermal mobility is lost and the atoms line up in billions to form large crystals. Such crystals represent the state of minimum energy for this system, and for slowly cooled systems, the nature is able to find this state. If a system of liquid metal is cooled to fast, it does not reach a minimum state but ends up in a polycrystalline or amorphous state having a somewhat higher energy state.

Thus, the essence of the process is slow cooling, allowing an optimal internal redistribution of the system while its elements loose mobility. This is the technical definition of *annealing*.

While cooling, a system's energy state might elevate in an attempt to find a new minimum energy state. But as the temperature drop, the likely hood of such hill-climbing moves decrease. If the cooling is sufficiently slow, the system will reach a steady state following the *Boltzmann* probability distribution.

This analogy is fairly close to the heuristic used in computer science and successfully describe its generality. The method has been used in a wide range of optimization problems, including NP-complete [7] problems such as the Traveling Sales Person (TSP).

Given the objective function f and the solution space Ω, define $N(\omega)$ to be the *neighborhood* function for $\omega \in \Omega$. All neighbors to a solution ω can be found immediately in SA.

SA starts with an initial solution ω. A neighboring candidate solution $\omega' \in N(\omega)$ is then generated following some scheme. The acceptance of the new candidate solution is based on the Metropolis [ea53] criterion which model the thermodynamic behavior described above. The candidate solution is accepted with the probability

$$P[\text{acceptance of } \omega' \text{ as the next solution}] = e^{\frac{-(f(\omega')-(f(\omega)))}{t_k}} \tag{3.1}$$

where t_k is the temperature at SA iteration k.

The probability of generating a candidate solution ω' from the neighbors of ω is $g_k(\omega, \omega')$ where

$$\sum_{\omega' \in N(\omega)} g_k(\omega, \omega') = 1, \text{ for all } \omega \in \Omega,\ k = 1, 2, ..., \tag{3.2}$$

This give the probability function

$$P_k(\omega, \omega') = \begin{cases} g_k(\omega, \omega') e^{\frac{-(f(\omega')-(f(\omega)))}{t_k}} & \omega' \in N(\omega) \\ 0 & \omega' \notin N(\omega) \end{cases} \tag{3.3}$$

[7] NP is a class of decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine. All NP problems are reducible to an NP-complete problem.

of the transition between state ω and ω' if $\omega \neq \omega'$.

The SA algorithm is outlined in Algorithm 3.

Algorithm 3 Simulated Annealing.

1: Select an initial solution $\omega \in \Omega$
2: Select a candidate solution generator, $N(\omega)$
3: Select a cooling schedule, $T(k)$
4: Select a repetition schedule, M_k, that defines the number of iterations executed at each temperature
5: **while** Stopping criterion is not met **do**
6: **for all** Iterations in M_k **do**
7: Generate a candidate $\omega' \in N(\omega)$
8: $\omega \leftarrow \omega'$ with probability $e^{\frac{-(f(\omega')-(f(\omega)))}{t_k}}$
9: **end for**
10: Lower the temperature according to $T(k)$
11: **end while**

The generality make simulated annealing a useful tool, however, there can be quite a lot of problem-dependant subtley in the term "sufficiently slowly" annealing. Success or failure is often determined by the choice of annealing schedule as well as the randomly generated candidate solutions decided by the properties of equation (3.3).

[Ing89] emphasize that to be sure that SA will statistically deliver a true global optimum, the cooling scheme must be slow enough that every system configuration can be sampled infinitely often in annealing-time (IOT). That is, the probability of not generating a state ω IOT yield 0.

Given the SA above, called Boltzmann Annealing (BA), the cooling must be

$$T(k) = \frac{t_0}{\ln k} \tag{3.4}$$

where t_0 is the start temperature and k the annealing-time steps. This is very slow, and has lead to other distributions like the *Cauchy* distribution for fast annealing (FA). FA will cool with

$$T(k) = \frac{t_0}{k} \tag{3.5}$$

which is exponentially faster than the previous schedule. An even faster SA method is proposed in [Ing89], very fast annealing, yielding a cooling schedule like

$$T(k) = t_0 e^{(-ck)} \tag{3.6}$$

However, fast and very fast annealing is not very attractive in practice because the ease of coding and implementing SA is lost [Ing93]. As a result, *simulated quenching* is similar to simulated annealing but use a temperature schedule too fast to satisfy the sufficiency conditions required to statistically find a true global optimum.

3.5 Measurement

Timing is used in two different contexts in this thesis:

1. The solution method described in 1.2 requires that we empirically measure the performance of a loop.

2. Measurements are needed in order to determine if the dynamic optimization actually result in faster execution.

Both items, and the first one in particular, put hard constraints on the accuracy of the measurements. The execution time of a loop may be extremely small, requiring a high resolution timer.

The second item require means to discriminate between two or more systems, this is also implicit in the first item, since we need to determine which combination of routines that are best among numerous possibilities. We do not want to claim that one solution is better than another on false premises.

The following sections will address the issues needed to consider for both of the above items.

3.5.1 Measuring Time

When timing the execution of a particular event, we want the timing results to cluster. High clustering is related to the high *precision* of the a timer rather than the *accuracy*. In the particular case of comparison between several alternatives on one system it is important with high precision. The accuracy, which relates to how close the timing results are to the actually time spent, is in fact irrelevant since we only measure the relative differences between two combinations of routine selections.

Timing Errors

There is always at least one error in timing results, and that is the effect of using the timer. The program statements added to a program to access the timer change the program we want to measure into something else. We cannot get rid of this effect, only try to make it insignificant.

It is very likely that random errors in a timing occur. Non-deterministic events, such as cache misses, system exceptions, memory page faults, and so on, will perturb the system. As a result, multiple measurements of the same event will most likely yield different results.

It is impossible to predict the precise effect of specific sources of random errors, but it is possible to develop a statistical model to describe their effect on the experimental results. Random errors are unbiased in that a random error has an equal probability of either increasing or decreasing a measurement compared to the true mean. We exploit the fact that the probability of obtaining a particular measurement is proportional to the number of paths that lead to that measurement. This produce a binomial distribution for the possible measurement outcomes, and as the number of paths increase, the binomial distribution will approach a Gaussian distribution centered around the true mean value [Lil00].

Measurement of Short Intervals

There are some difficulties related to the issue of timing intervals which have a very short execution time relative to the resolution of the timer.

The finite resolution of a timer introduces a random quantization error into all measurements made using the timer. The actual event time is rarely a perfect multiple of the timer resolution, so the time reported is usually a rounded result. This rounding is random, unpredictable and impossible to control. As a result, measurements of intervals whose duration is of the same magnitude as the timer resolution is questionable. As a rule of thumb, the event duration should be at least two to three orders of magnitude larger than the timer resolution [Lil00].

It is possible, of course, to measure intervals whose duration is smaller than the timer resolution. We can use the fact that the outcome of a measurement of such an interval is one of two; either one or zero clock ticks are reported. This is a Bernoulli process with the following properties:

1. The experiment consists of n repeated trials.
2. Each trial results in an outcome that may be classified as a success or a failure.
3. The probability of success, denoted by p, remains constant from trial to trial.
4. The repeated trials are independent.

When the probability p is known after n separate trials, it is straightforward to calculate the duration of the measured interval. The fourth property of the Bernoulli process is most likely not fulfilled since there are typically dependencies between the trials.

3.5.2 Difference Between Systems

Both items in the beginning of section 3.5 require methods to determine whether one option is better or worse than another. The measurement errors in section 3.5.1 indicate that this is not a trivial task.

Confidence Intervals

The timing result of a system is shown to have a normal distribution. Instead of finding a point estimate for the mean value, we use an interval estimate. Through a sample from a population we can compute a $(1-\alpha)100\%$ *confidence interval*.

We can establish a confidence interval with a certain confidence coefficient for the timing of an event. Further, we can use several confidence intervals to determine whether there is a significant difference in the time used by two events. If the confidence intervals for two alternatives do overlap, then it is impossible to say that any differences seen in the mean value are not due to random fluctuations. If they do not overlap, however, we conclude that there is no evidence to suggest that there is not a statistically significant difference [Lil00].

Confidence interval of the difference between two means, $\mu_1 - \mu_2$, is found by:

$$(\overline{x}_1 - \overline{x}_2) - t_{\alpha/2}\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} < \mu_1 - \mu_2 < (\overline{x}_1 - \overline{x}_2) + t_{\alpha/2}\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \tag{3.7}$$

where $\overline{x}_1$ and s_1^2, and $\overline{x}_2$ and s_2^2 are the means and variances of independent samples of size n_1 and n_2 respectively. t_α is a critical value given by the t-distribution[8], and can be looked up in tables in [Lil00] and [WMM02].

If this interval contain 0 there is no statistical significant difference between the two alternatives.

The confidence-interval approach for comparing two alternatives is quick, simple, and intuitively satisfying.

ANOVA

Analysis of variance (ANOVA) is a robust and general technique for measuring the difference between alternatives. It divides the total variation of measurement results into two separate components: one for random fluctuations within one alternative, and one for the differences between the alternatives.

The method is used in this work to effectively determine the effects of the dynamic optimization. A more detailed description can be found in Appendix A.

[8] The correct *degrees of freedom* are found by

$$v = \frac{(s_1^2/n_1 + s_2^2/n_2)^2}{[(s_1^2/n_1)^2/(n_1 - 1)] + [(s_2^2/n_2)^2/(n_2 - 1)]}$$

Chapter 4

Implementation

This chapter present a high level description of the implementation of the solution method proposed in 1.2, utilizing much of the theory presented in the previous chapter. The actual code is described in appendix B.

First, the modifications necessary to Scali MPI Connect (SMC), are presented. Then a description of the loop detection follows, including a discussion of issues that came up during the implementation. Finally, details of the implementation of the simulated annealing are given.

4.1 Scali MPI Connect

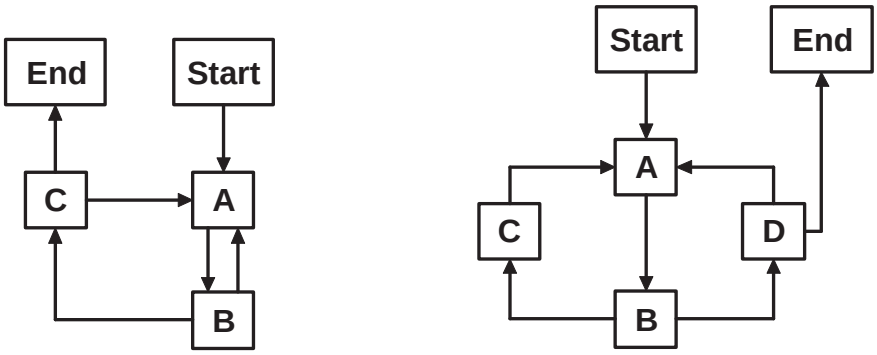
The solution method imply two necessary modifications to SMC.

1. Means to record return addresses.
2. Functionality to detect loops and perform global optimization.

The first point is obtained by inserting a small block of code into all MPI function calls, including the FORTRAN wrapper functions in the MPI implementation. The block of code detect the return address, and some of the function characteristics such as the message size and whether it is a send or receive primitive. This information is sent to the functions of the second point which does not need any other interaction with the MPI implementation than means to manipulate the use of intra-copy routines. There is already a built-in function in SMC for this purpose.

4.2 Return Address

The call stack stores information about the active subroutines of an application. More specific, it record information about which address to return control when the execution of the current routine is finished. Hence, by inspecting the call stack, we can find the specific location from which the current routine has been called. The call stack is accessible through assembly instructions or by usage of gcc compiler built-in. In particular, the routine $\_\_builtin\_return\_address\_(0)$ in gcc will return the memory address. The routine cannot be safely used to find more levels of return addresses [StGDC03].



(a) Two loops where one is contained within the other.

(b) Two loops with shared start.

Figure 4.1: Examples of node sharing between loops.

Execinfo is a part of the C library, primary used for debugging by programs like gdb[1]. It can obtain a backtrace of the current threads as a list of pointers and later translate these pointers to a list of strings. This can be used to find return addresses of deeply nested routine calls, but in practice it add a noticeable overhead which make it unsuited for dynamic loop optimization.

4.3 Loop Detection

The background theory of loop detection is covered by section 3.3, where algorithm 1 is presented. This algorithm is implemented as part of this thesis. There are some issues that come up when loop detection is used for timing and optimization.

4.3.1 Problems with Loops

Loops have some properties which are not desirable for timing and global optimization.

Node Sharing

Loops may share node elements. At run time, the variable setting associated with a node element will be determined by measurements of the loops. If a node is shared, it may obtain a different variable settings in different contexts. This is desired behavior. But when loops share nodes from the start of the loop until some point, or if one loop is fully contained within another, there is no way of telling which loop that is executed at run-time when at the entry node.

Figure 4.1(a) display two loops which share a common entry node. If the wrong loop is assumed to be executed, associated variable settings may become erroneous.

[1]The GNU Project Debugger. http://www.gnu.org/software/gdb/

Irreducibility

Nodes can also be shared at the end or in the middle of loops. This is not critical to determine which loop that is executed, but it lead to irreducible loops as described in section 3.3.3. These loops cannot be used for detection, since there is no knowledge of how the irreducible loop will be executed.

The solution to node sharing is to break the loops into disjoint intervals. We loose some information about the loop structure and the routine selection might not be optimal for the interval nodes. But in return, we get a practically feasible solution.

4.3.2 Intervals

From a set of loops we can directly obtain a set of disjoint intervals. Also, we need a set of "stop-addresses" which indicate the end of one interval, and possibly the start of a new one. These are necessary in order to time intervals correctly. We want to time the entire interval including effect of the last routine selection, but an interval may very well end in different MPI-calls. Algorithm 4 describe the procedure implemented to obtain the set of intervals induced by a set of loops.

Algorithm 4 Create disjoint intervals from a set of loops.

1: $T \leftarrow$ Set of all loop targets
2: $I \leftarrow null$
3: **for all** Loops $L = \{T_l, n_1, n_2, ..., n_b\}$ **do**
4: **if** $n_j \in T$ **then**
5: Split loop L into intervals $I_1 = \{T_l, n_1, ..., n_{j-1}\}$ and $I_2 = \{n_j, n_{j+1}, ..., n_b\}$
6: $I \leftarrow I \cup I_1 \cup I_2$
7: **end if**
8: **end for**
9: **for all** I_i **do**
10: **for all** $I_j > I_i$ **do**
11: $I_{intersection} = I_i \cap I_j$
12: $I_j \leftarrow I_j - I_{intersection}$
13: $I_i \leftarrow I_i - I_{intersection}$
14: $I \leftarrow I \cup I_{intersection}$
15: **end for**
16: **end for**
17: **for all** $I_i = \{T_i, n_1, ..., n_b\}$ **do**
18: Stopaddresses $I_i \leftarrow$ all destination nodes $n_b \rightarrow m_1, ..., n_b \rightarrow m_k$
19: **end for**

Given the program in algorithm 5, two loops will be detected, namely

$$L_1 = MPI\_Send, MPI\_Recv$$
$$L_2 = MPI\_Send, MPI\_Recv, MPI\_Allreduce$$

L_1 is completely contained within L_2, equal to Figure 4.1(a). Thus, L_1 will lead to an interval, I_1, which will have the stop-addresses at $MPI\_Allreduce$ and MPI_Send, and with the entry node

Algorithm 5 Interval with multiple ending

```
1: MPI_Init
2: Do some initialization
3: for i = 0; i < 10; i++ do
4:    for j = 0; j < 4; j++ do
5:       Do some work
6:       MPI_Send
7:       MPI_Recv
8:    end for
9:    Do more work
10:   MPI_Allreduce
11: end for
12: MPI_Finalize
```

MPI_Send. The other loop, L_2 will be reduced to an interval which only contain $MPI\_Allreduce$ with MPI_Send as a stop-address.

The two separate stop addresses for the interval induced by loop L_1 calls for two separate timing values. We start the timer when MPI_Send is called, and stop it as soon as one of the routines at the stop-addresses is called. It is essential that the impact of the memory copy routine used in MPI_Recv has on subsequent work is measured. It may very well be that the choice of memory copy routine will be different for the two stop-addresses, but as pointed out earlier, there is no way of telling in advance which stop-address that will be executed.

In this particular example $MPI\_Allreduce$ is the stop-address of I_1 10 times, while MPI_Send is stop-address for I_1 30 times and I_2 9 times. It is essential to utilize both stop-addresses of I_1 to get as many timing results as possible.

Figure 5.2 show the flowgraph of the NAS-CG benchmark. This is an extended example of the need to use intervals instead of loops. The coloring represent the intervals. The green nodes are a loop while the yellow, green and blue nodes are shared between two loops with different endings. If only shared-nothing loops were recorded, only the green nodes would be optimized.

4.3.3 Dynamic Loop Detection

Section 3.3.4 describe how to detect loops at run-time. There are, however, some issues to consider when implementing this.

1. At least one iteration of a loop is needed before it is detected.
2. All node information must be stored and re-checked during loop execution. In case of loops with shared nodes, we must know which loop that is executed, hence the re-checking. This lead to the next point.
3. We need to get confidence to the execution of loops with shared nodes, so we can speculate in which loop that is executed next.
4. The needed confidence reduce the number of iterations for SA.

5. At least one loop termination is needed before an external stop-addresses is recorded.
6. There are more control structures needed to maintain loop information than if the loops where known in advance. For instance, a table is needed to hold information about the signature of all MPI function calls seen during the current execution. This table would obviously become expensive in terms of memory usage and look-up speed as the number of differing calls increase.

An advantage for the dynamic detection is that it can correctly determine the execution of an irreducible loop due to the confidence requirement. Algorithm 6 describe the implementation of dynamic loop detection. It will correctly identify intervals, so routine selection in outer loops will not be affected by the routine selection in inner loops.

Algorithm 6 Dynamic loop detection.

1: $s \leftarrow$ signature from MPI call, including return address.
2: $n_s \leftarrow lookup(s)$ lookup a node which represent that particular communication call.
3: **if** $n_{previous}$ is a branch B and n_s is a stop-address **then**
4: Perform simulated quenching on the loop previously executed.
5: **end if**
6: **if** s has been found earlier **then**
7: **if** $next(n_{previous}) \neq n_s$ **then**
8: We are out of a loop, n_s may be a target for another. Check for a loop with n_s as target, if not found, iterate through all nodes n starting with n_s where $previous(next(n)) = n$. If already found, add n_s to the stop-addresses of that loop.
9: **end if**
10: **end if**

4.3.4 Static Loop Detection

Static loop detection require a "dry run" of an application were a flowgraph is constructed. This flowgraph is used as input to our implementation of Algorithm 1, which in turn produce input to Algorithm 4. The derived intervals are further used as input to the MPI implementation when run with the particular application. Through multiple executions, the MPI library is iteratively optimized to suit the application.

Shortcomings of Algorithm 1

There are some issues in the algorithm described in [SGL96]. Two of them has already been addressed, namely the irreducible loops, and if nodes are shared between two loops.

The former cannot be correctly resolved, but based on empiric data such a flowgraph with weighing, or through confidence in loops as in dynamic loop detection, we can speculate how an irreducible loop will be executed.

The latter issue is not addressed at all in the article, but the algorithm described there, do collapse the correct nodes. The issue emerge at step 11 in algorithm 1. We can identify two problems:

1. When two loops share the first n nodes, but then diverge, the loops should be recorded separately before they are collapsed into the entry node.

2. When a node is a target for two or more loops, and one of them is contained within the other, as in the program of Algorithm 5, the contained loop must be collapsed first. This is correctly executed, but the early recording of loops in the above point return erroneous loops.

Hence, for each incoming BJ edge $m \rightarrow n$ we need to record a new set of nodes which can reach m without going through n. If one set is a subset of another, we have found a loop that is fully contained.

4.4 MPI Calls

All point to point calls known from section 3.1.1 are suited for this optimization. The collective routines are not used since we do not know if all processes included in the communicator will discover equal intervals. Likewise, $MPI\_Barrier$ is not used within a point to point call to increase measurement accuracy, since all the processes in that communicator then has to be able to pair their communication calls.

The collective operation functions, along with the MPI_Wait family, are used in the loop detection and in intervals. This is to use them as stop-addresses and possible reduce the execution time of intervals.

4.5 Simulated Annealing

Instead of the simulated annealing from section 3.4.1, we use simulated quenching. The run-time conditions does not suit the slow convergence of simulated annealing. Even fast and very fast simulated annealing, which do converge quite rapidly, is not appropriate because they require much more instructions to be executed.

Simulated quenching process multiple solutions at every temperature. Similarly, we evaluate the performance of an interval and its configuration for each time it is executed. When a number of iterations has been executed, the temperature is lowered according to a geometric or exponential cooling scheme like the one in equation (3.6) from Section 3.4.1.

If an interval consist of many communication calls, consequently having a large solution-space, it should be evaluated correspondingly many times. However, this is not typical behavior of program code, as larger loops tend to have less iterations than smaller ones. We want to maximize the utilization of a small amount of iterations by recording the results of various system configurations.

This coincide with the fact that there is no correlation between the deviation of two solutions in terms of the empiric measurement result and the routine selections. A nearby neighbor created by a small deviation of the previously accepted solution may result in a huge difference in the time spent executing the interval. Equivalently, a distant neighbor is unpredictable since the solution-space is very “bumpy”.

This is illustrated in Figure 4.2 where we see the solution space of a single communication pair MPI_Send - MPI_Recv. If there were two such pairs in one loop, the solution space would be squared in the size of combination numbers.

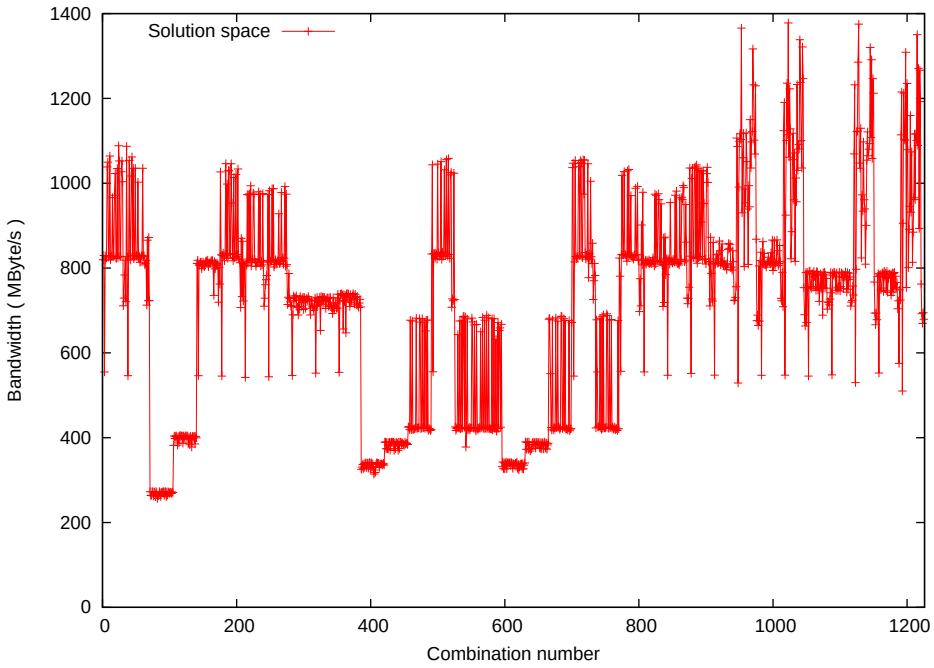


Figure 4.2: A solution-space of a MPI_Send - MPI_Recv pair.

By maintaining a heap of timing results, all information about the performance of a routine in a particular communication call is not lost. Instead, we get an indication on how it will perform if selected later and seemingly useless solutions can be avoided in the future, thus reducing the number of iterations necessary. This strategy may very well exclude a global optimum, and definitively do not fulfill the requirements needed to assure that the algorithm statistically will find a global optimum. However, it is a practically feasible solution.

4.5.1 Reducing the Solution Space

An interval consisting of n communication calls which can choose among D variables, result in a potentially large solution space with a total of D^n unique system configurations. In particular, an interval consisting of two send and two receive calls will have a solution space of $35^4 \approx 1.5 * 10^6$ elements. We want to reduce this solution space without loosing a potential global optimum. There are some routines that can safely be excluded based on:

Message size This apply to several situations:

- The message is very small. There is no need to do any selection in this case, since the general overhead of the message passing, the simulated quenching and all other instructions carried out in the interval will make the differences in intra-copy routines insignificant.
- The message is average sized. Memory aligned and messages whose length is a multiple of a power of two is fastest copied with routines which use MMX and SSE instructions. Generally, backward copying routines will outperform forward copying routines since they avoid any cost from the hardware prefetcher.

- Large messages. Blockcopying routines which utilize SSE instructions and TLB priming in addition to prefetching are likely to outperform other routines.

Non-continuous data These messages are packed into a continuous memory location at sender side, and reversely unpacked at the receiver side. This process involve multiple memory copy calls that will give cache effects on the subsequent instructions executed. Routines that are suited for large messages may not perform as well when the message data is non-continuous.

Use of derived datatypes will introduce multiple memory copy operations with various characteristics. Given our solution method, it is not possible to set which routine that should be used of each of these operations. A more flexible memory copy operation is needed, like the default in Scali MPI Connect which is a collection of operations each specialized to operate on different data sizes.

The need for flexible memory copy operations is also evident in collective operations and calls to the $wait$ family of functions. As a consequence, the default operation, $scamemcpy$, is included.

4.5.2 Convergence Properties

As stated above, the simulated quenching is implemented somewhat different from the original Boltzmann annealing. We use the same acceptance probability, but with a different temperature scheme and a weighted selection of candidate solutions.

In addition, we have the possibility to "restart" the annealing process if the current system configuration is significantly worse than the best result found. This is a common technique called *re annealing*.

Binary Heap

The timing result for each time an interval has been completed is recorded. This result is compared to the best ever result and rated accordingly. For each communication call, there is a binary heap structure maintaining the relative performance of all the intra-copy routines at that particular communication call.

In order to avoid unfair rating due to timing errors, only relative improvements are recorded. The first loop iteration is also not timed to avoid possible start up penalties. Consequently, this will lead to timing the best result instead of averages, which was recommended in Sections 3.5.1 and 3.5.2. This can be justified by the fact that the measurement of small intervals are likely to be averaged out since they are at the same order of magnitude as the timer, and thus require several iterations before a valid timing result can be obtained. For large intervals, it is likely that only the best result are recorded.

The confidence interval approach of Section 3.5.2 were found to be too expensive to compare two solutions since it greatly affected the number of SA invocations. Relative SA convergence speed were good, but could not make up for the number of iterations needed to obtain a good level of confidence.

Candidate Generation Criteria

The neighborhood function $N(\omega)$ of section 3.4.1 highly influence the efficiency of simulated annealing [Mos93]. The function can enforce a topology and thus "smoothing" the solution space. [Fox93]

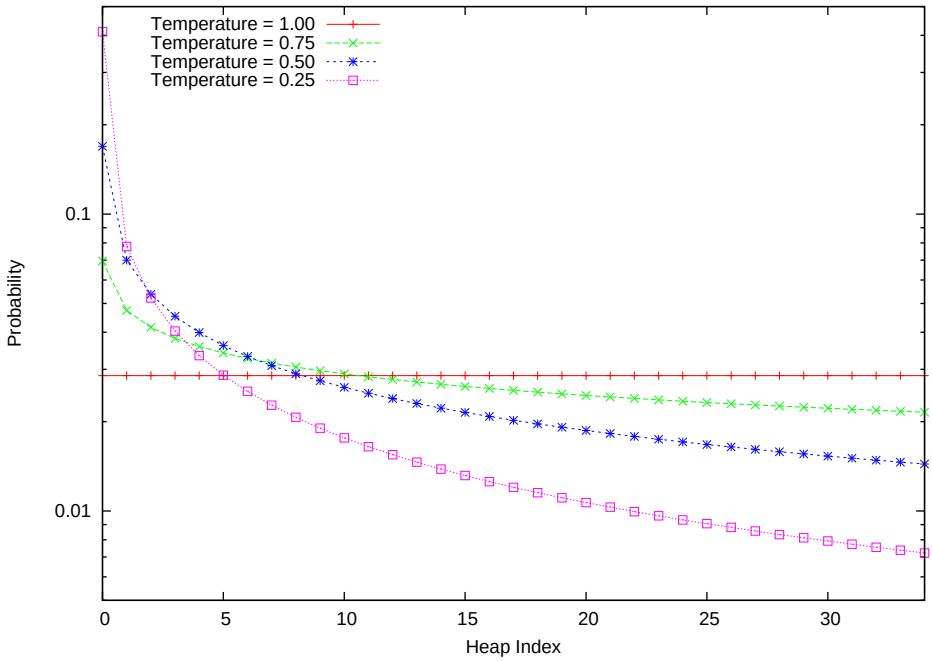


Figure 4.3: Selection probability

suggests that instead of blindly generating neighbors, we should develop an intelligent generation mechanism that modifies the neighborhood and its probability distribution to accommodate search intensification or diversification.

We do this through the binary heap structure by modifying the probability function $g_k(\omega, \omega')$ of selecting a candidate solution ω'. Probability distribution generated by one of the functions implemented is illustrated in figure 4.3. As the temperature cool, it is less likely to select a routine whose performance has been poor in the past. At start up, all routines are equally likely to be selected.

This is close to a *probabilistic tabu search* which is a meta-heuristic to help guide SA. In probabilistic tabu search, the probabilities of generating and accepting each candidate solution are set as functions of both a temperature parameter (as for SA) and information gained in previous iterations (as for tabu search) [HJJ03].

Stopping Criteria

The simulated quenching is stopped after the interval has been evaluated a number of times. The size of the state-space decide the amount of iterations needed at each temperature. Also, if the number of solution changes that result in a best time exceed a threshold the temperature is prematurely reduced. This is because SA is exploring a very favorable solution-valley, so we can reduce the number of evaluations.

Chapter 5

Benchmarks and Testing

There are many factors to take into consideration when performance testing the effect of a change in parallel software. This chapter will give a characterization of the benchmarks used, as well as the system properties and known causes of measurement deviations and how to handle them.

5.1 Benchmark Programs and Characteristics

The benchmark programs used are vital in the sense that we want to investigate various effects of our solution method. They should have the following properties.

1. Contain loops of point-to-point communication calls. This is essential for our solution method. Actually, it is almost impossible to find benchmark programs that do not fulfill this requirement.
2. They should mimic the communication and computation pattern of regular parallel applications.
3. The number of processors used should be at least two and preferably, the benchmark program should be able to employ a variable number of processors.
4. The amount of data transmitted per communication call should be tunable. This may also increase the amount of non-communication work done within the loops. Ideally, these two components should be controlled separately.
5. Variance between immediate and late use of data before/after a send/receive primitive and use of derived datatypes to investigate the effects of routine selection.

In addition, we would like to benchmark the effects of the global optimization heuristic. Both in added instruction overhead, and its ability to reach an optimum.

Naturally, one benchmark program can not satisfy all the above requirements. We have utilized several different synthetic benchmark programs, no real applications. In particular, we use two kernels of the *NAS parallel benchmark*, a point-to-point bandwidth benchmark, and a simple benchmark that mimic the communication pattern of a two dimensional partial differential equation solver using a five point stencil.

Only benchmarks with deterministic control flow is used. Programs with non-deterministic behavior are a interesting case, but non-determinism in control flow may also lead to non-determinism in execution time. The effects of measurement errors are likely to have a greater impact on such programs. For instance, a bad load balance schedule can skew the results in a much greater extent than any selection of intra-copy routines.

5.1.1 NAS Parallel Benchmarks

The NAS parallel benchmarks are a set of programs designed as a part of the NASA Numerical Aerodynamic Simulation (NAS) program originally to evaluate supercomputers. They have the communication and computation characteristics of large-scale computations in real-world applications. The benchmark suite consists of five kernels (EP, MG, FT, CG, IS) and three pseudo applications (LU, SP, BT) programs.

NAS-CG and NAS-MG kernels use blocking and immediate point-to-point message passing, while the other benchmarks rely on collective communications. Both the kernels can be run with a various number of processors and with three different problem classes, providing numerous combinations of computation costs and message sizes.

Both benchmarks used are from the NPB2.3 [BBB$^+$91] suite. The information of the NAS-CG and NAS-MG kernels are derived from code inspection, [BBB$^+$91] and [TKN$^+$].

NAS-MG Benchmark

The NAS-MG multigrid benchmark solves Poisson's equation in 3D using a multigrid V-cycle. The multigrid benchmark carries out computation at a series of levels and each level of the V-cycle defines a grid at a successively coarser resolution. This implementation from NAS is said to approximate the performance a typical user can expect for a portable parallel program on a distributed memory computer [BBB$^+$91].

The communication is highly structured and goes through a fixed sequence of regular patterns. For each level of the V-cycle, there are periodic updates of the boarder regions of a rectangular data volume from the neighboring processors in each of the six spatial directions. For each of the three spatial axes, two messages (except for the corner cases) are sent using point-to-point communication to update the border regions on the left and right neighbors.

Figure 5.1 illustrate a simplified version of the flowgraph of the NAS-MG kernel using four processors. There are three small reducible loops in this benchmark, however, the figure does not show that these loops are executed many times with a different message size for each level of the V-cycle. Thus, a loop for each message size is recorded, giving a total of 60 loops in the class B problem size.

NAS-CG benchmark

In the NAS-CG benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel benchmark test irregular long distance communications and employ sparse matrix vector multiplication.

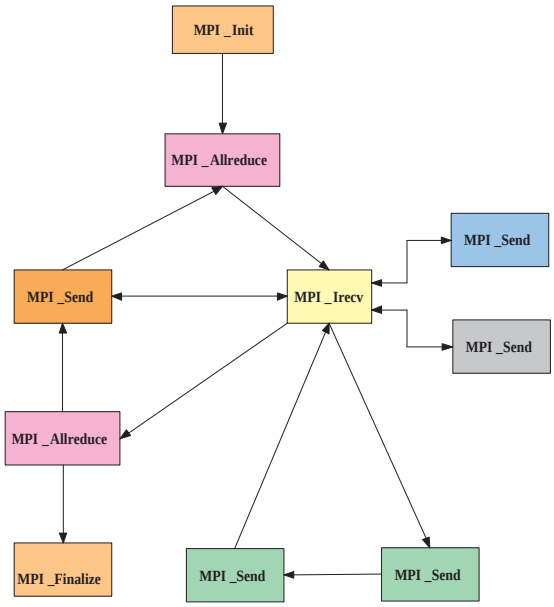


Figure 5.1: NAS-MG flowgraph

The NAS-CG benchmark involve multiple iterations of a solution to the system of linear equations, $Az = x$, using the conjugate gradient method. It compute the residual norm at the end of each of these iterations. After each iteration, the eigenvalue is estimated with a shift λ. The size of the system, number of iterations involved and shift applied to the eigenvalue estimate is determined as a part of the initial setup of each problem class.

The procedures that involve most of the communication are matrix vector multiplication, vector dot product and computation of the residual norm. The computation of the matrix vector product involves a recursive doubling based on pairwise exchange, while the calculation of the residual norm involves a recursive reduction process. This is implemented using point-to-point communication, yielding an implicit barrier operation.

Figure 5.2 show a simplified version of the NAS-CG flowgraph using four processors. There are three reducible loops in this flowgraph. The green nodes form an inner loop contained within two other loops which share their first nodes. The coloring of the nodes illustrate the intervals that these loops are reduced to. The number of iterations and the message sizes are decided by the problem class. For problem class B, the inner loop is executed 75 times while the outer loops are executed 15 and 4 times.

5.1.2 Other benchmarks

The two NAS parallel benchmarks fulfill the first four requirements above. However, the communication to calculation ratio are low. Hence, they mimic real world applications but do not emphasize small changes within MPI implementations. Thus, two other synthetic benchmark programs have been used.

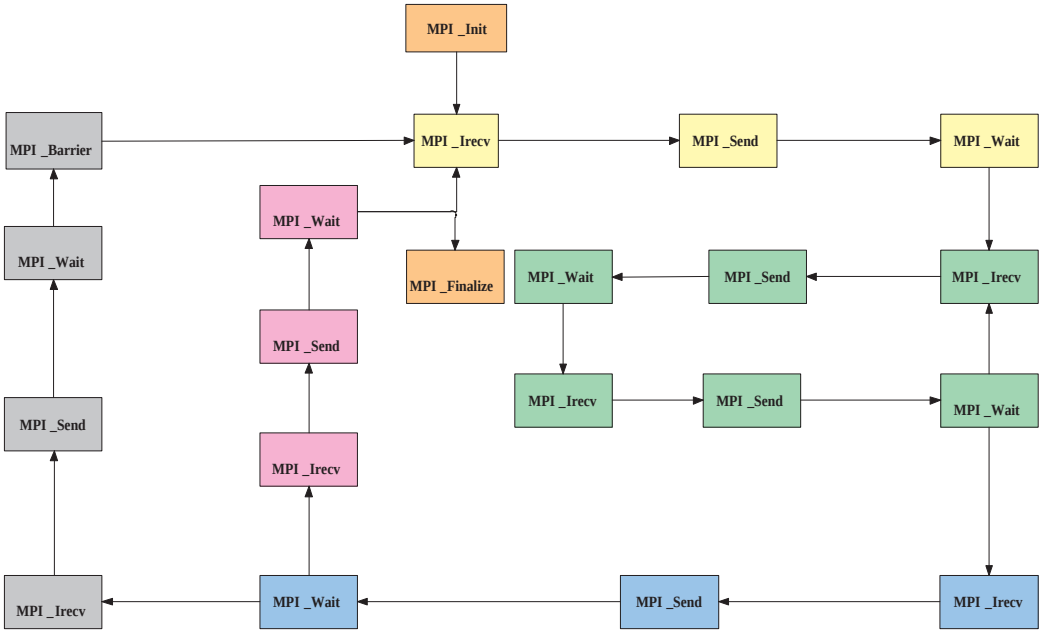


Figure 5.2: NAS-CG flowgraph

Bandwidth Benchmark

The bandwidth benchmark is as simple as possible. The essence of the benchmark is a loop containing a point-to-point communication call, typically a MPI_Send - MPI_Recv pair. No real work is done in the iterations, but use of derived datatypes or invocations of the standard functions $memset$ or $memcmp$ may skew the optimal routine selection and satisfy the fifth requirement above. Measurements with this benchmark will underline the effects of the internal changes in the MPI implementation since there are no other calculations performed.

A modified version of the *bandwidth* example program from Scali MPI Connect is used.

Cartesian Grid Benchmark

This benchmark will mimic the communication pattern of 2D PDE solvers that use a five point stencil. Given an rectangular grid, each process need to communicate with logical neighbors in all four directions. This benchmark is similar to the bandwidth benchmark in that it does not perform any useful calculations, but it is able to utilize more processors and have a slightly different communication pattern.

5.2 System

Our solution method as a concept is independent of the underlying hardware, but the memory copy routines used are not. Two systems were used to run the benchmark:

1. An SMP with four 1.8 GHz AMD Opteron processors with ethernet, myrinet and infiniband interconnect as well as shared memory communication.
2. A cluster with 15 dual-core Pentium III processor nodes with ethernet interconnect.

Different interconnects will yield different transfer time and other variances and standard deviations for the message passing, ultimately affecting the level of confidence the routine selection can operate on and the convergence rate of the global optimization heuristic. Ideally, the transfer time should be long enough to satisfy the timer requirements, see section 3.5.1, with a minimal variance. As a consequence, most benchmarks were performed using the shared memory communication on the SMP, since this resulted in least measurement deviation.

5.3 Determinism

There are many elements that may interfere with the benchmarks. This section identify known sources of deviation and how to handle them.

Processor Allocation

When starting an MPI-application the processes will map to the processors available. It is crucial that this mapping will persist throughout the execution. Any suspension or migration may cause cache reinitialization, causing a potentially large performance penalty. In addition, this mapping must be reproduced every time the application is executed for timing purposes.

Unfortunately, there are no means (yet) in Scali MPI Connect to lock a process to a processor within an SMP. However, all the benchmarks were run without any background load, so there were no process suspension or migration.

Interconnect Traffic

There is no possibility of knowing the effects of interfering interconnect traffic. Other jobs running concurrently on the benchmark system may reduce both bandwidth and response times between the processors allocated to the benchmark program.

As stated above, all benchmarks were run without any background load, so there was no deviation due to interconnect traffic.

Timer Resolution

The strict timer resolution requirements for loop timing is achieved by utilizing high precision hardware timers. The benchmarks use MPI_Wtime which, in fact, utilize the same timer as the loop timing code.

Hardware Interrupts

There is uncertainty about the impact of this. If an arbitrary processor is chosen to handle an interrupt, it may skew the timing result. It may cause the cache to be flushed.

Randomized Search Heuristic

There is quite a bit of randomization in the simulated annealing heuristic. This randomization lead to various convergence and found solutions and thus to non-deterministic timing properties.

The last two causes of deviation are identified and quantified by the statistic method of ANOVA, see Section 3.5.2 and Appendix A, hence, we can trust the measurement results obtained.

Chapter 6

Results and Discussion

This chapter present the results obtained while executing the benchmarks described in the previous chapter. First, we find the performance increase possible using point-to-point communication with optimal selection of intra-copy routines. Then our solution method is justified through benchmarks which illustrate the importance of run-time optimization. Later, the convergence speed and introduced overhead of our solution method is found, and finally it is applied on the NAS parallel benchmarks.

All benchmarks except for the NAS-CG and NAS-MG were run only on the four processor SMP with shared memory communication. This system introduce very little variance into the benchmark results and emphasize the effects from our solution method. The number of intra-copy routines to select from were all 35 unless otherwise is stated.

6.1 Bandwidth Benchmarks

It is obvious that any performance increase made by our solution method depend on the difference between the optimal selection and the default selection in Scali MPI Connect. We present here the bandwidth obtained with point-to-point communication as a function of the amount of data transmitted.

Figures 6.1 and 6.2 shows the bandwidth obtained by a "ping-ping" communication pattern. This communication consists of one MPI_Send - MPI_Recv pair. The lines in these graphs have the following interpretations:

Asymmetric : best possible combination of intra-copy routines.
Symmetric : best intra-copy routine when equal at the sender and receiver side.
Standard in SMC: the default selection in Scali MPI Connect.

Figures 6.3 and 6.4 displays the results for the "ping-pong" communication pattern. In this pattern, a message is send back and forth between a pair of processes, using two pairs of MPI_Send - MPI_Recv.

These figures show how a potential performance improvement vary the communication pattern, the message size, and when the data is referenced.

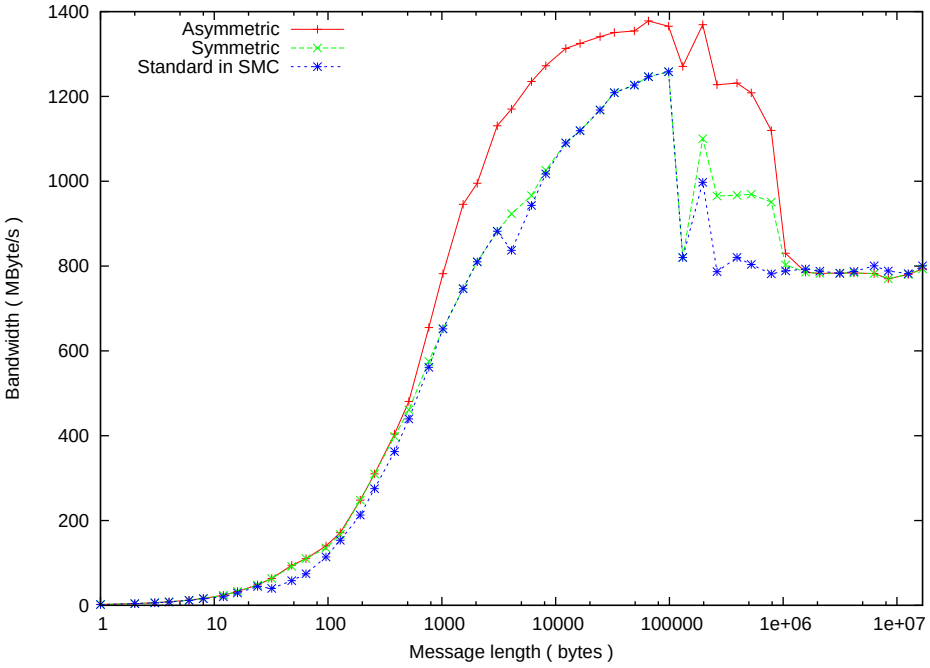


Figure 6.1: Ping-ping

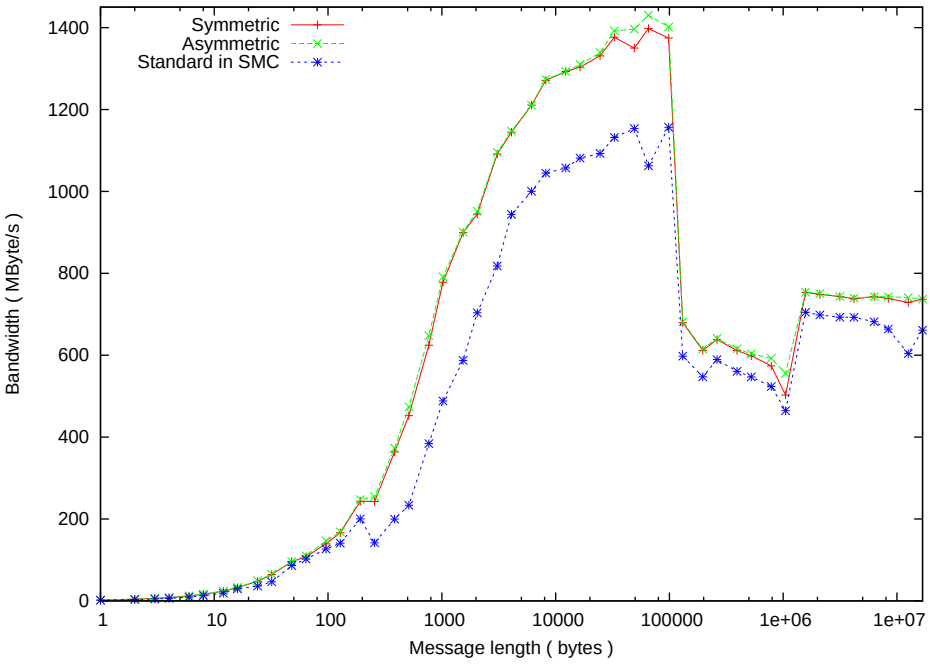


Figure 6.2: Ping-ping, touch message buffer

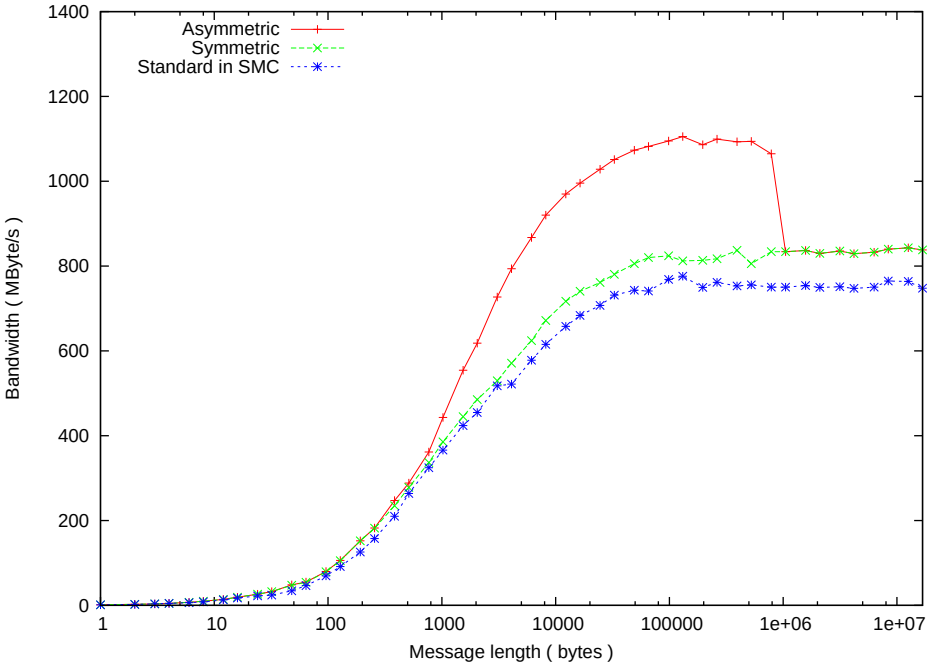


Figure 6.3: Ping-pong

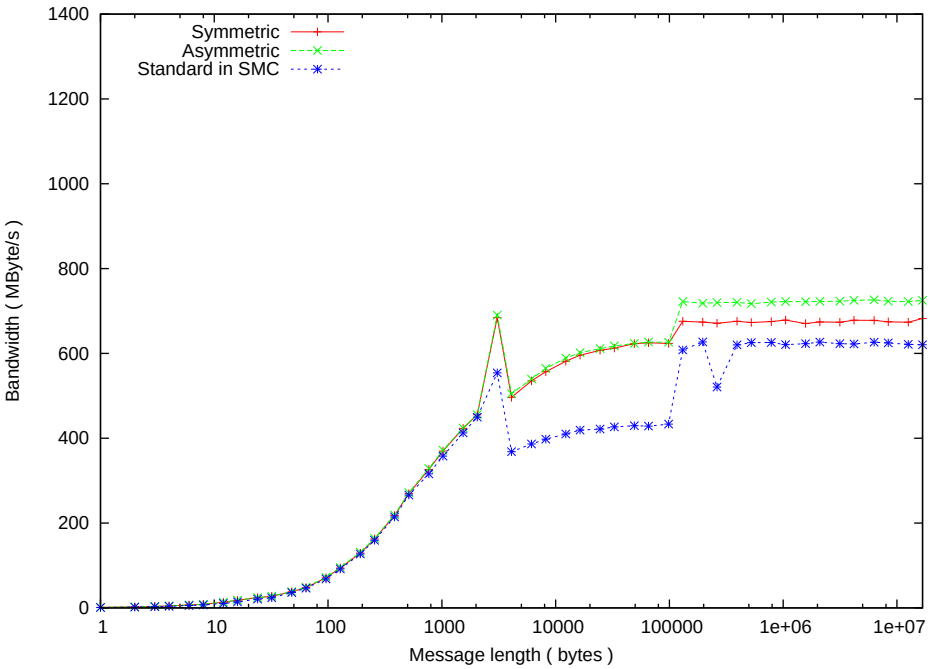


Figure 6.4: Ping-pong, touch message buffer

6.2 Data Dependencies

One of the arguments to use loop detection and run-time simulated annealing instead of a benchmarking strategy such as in [FY05] (see related work Chapter 2) is the fact that program characteristics such as data layout and access patterns will not be optimized with that strategy. This information about a program is only available at run-time, hence the loop performance timing is ideal for this purpose.

Two data dependency factors are evident.

1. Use of derived datatypes may cause unstructured data layout and lead to multiple memory copy operations before and after communication calls. This will affect the routine selection at both sender and receive side.
2. The access pattern of the data that are communicated. If referenced immediately before or after a communication call, the intra copy routine used should exploit the fact that the data are resided or should be resided in the cache hierarchy.

These dependencies were tested with the benchmarks described in Section 5.1.2

6.2.1 Derived Datatypes

We use the Cartesian grid benchmark which imply an MPI_Type_vector datatype with regular strided data rather than a user-defined datatype with irregular data layout. The main purpose of this particular experiment is to prove that the best routine selection when communicating non-continuous data will differ from the selection when transferring continuous data of equal total sizes. Hence, MPI_Type_vector will suffice.

The number of memory copy routines available were heavily reduced in order to obtain a solution space that could be fully explored. Figure 6.5 show the solution space with continuous and non-continuous data layout. Each of the combination numbers refer to a combination of intra-copy routines used. There were a total of 512 possible combinations in this benchmark.

From the figure we can conclude that the optimal routine selection depend on the layout of the data to be transmitted. The best performance for continuous data is shown to be at combination 147, which is clearly not the best result for non-continuous data (which is 269).

6.2.2 Data Access Pattern

Equal to Section 6.2.1 it suffice to show that the best selection of intra-copy routines will vary with when the communicated data are referenced. We use the bandwidth benchmark, comparing the difference in optimal routine selection when touching the data immediately before and after a transfer and not referencing the data at all.

Figure 6.6 displays two solution spaces. One of them touch the data transmitted right before and after the communication call, while the other does not. It is clear that the optimal solutions differ, hence an optimal selection of copy routine has to be determined at run-time. Another interesting effect seen

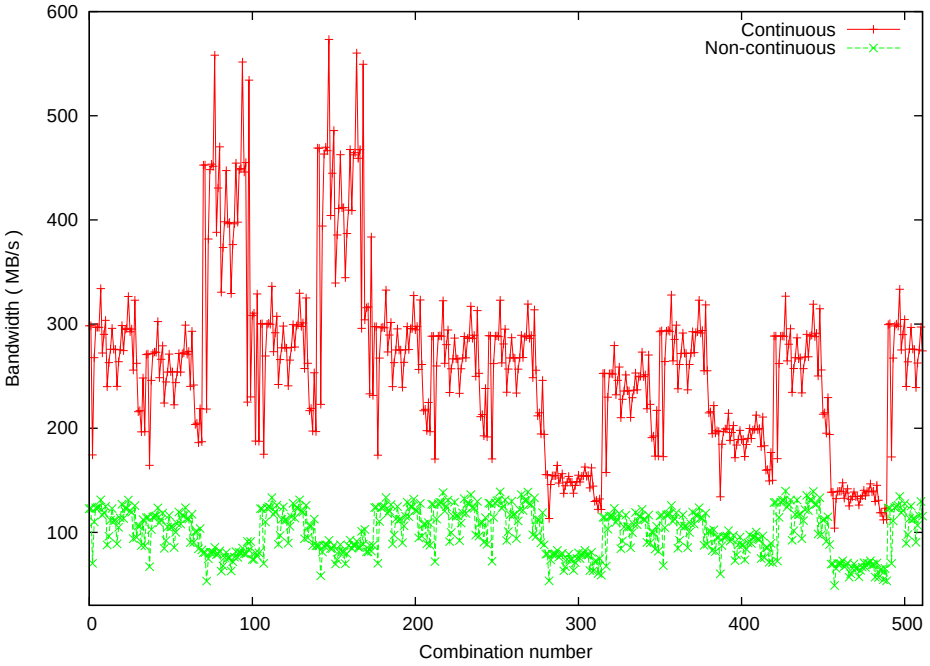


Figure 6.5: Two solution spaces. Communicating continuous and non-continuous data.

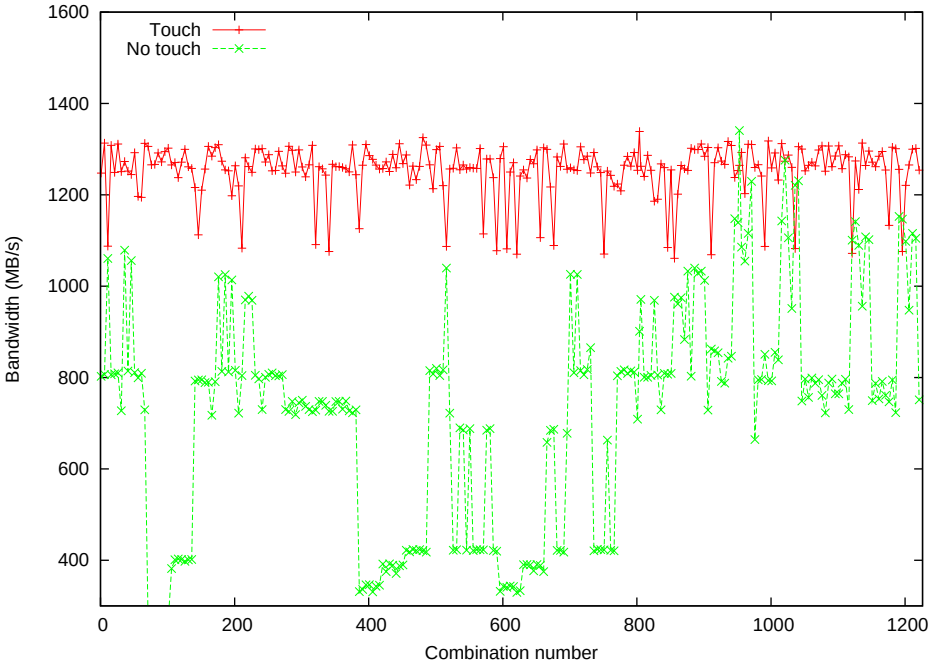


Figure 6.6: Two solution spaces. No use of transmitted data, and touching transmitted data.

is that the touched data causes the bandwidth to be increased. This solution space is for a relatively small message size, so the cache effect obtained by touching the data is very evident.

6.3 Simulated Annealing Characteristics

We want to assess the direct effects of the simulated annealing (SA). In particular, the rate of convergence and the introduced overhead of the SA implementation are investigated.

To isolate these effects, we use the bandwidth benchmark described in Section 5.1.2.

A slow convergence rate will make SA useless to optimize applications that are not loop and communication intensive. Also, the extra calculations of SA cannot imply a time penalty of the same order of magnitude as the loop it is optimizing since it could potentially skew the solution selection of SA. If the convergence is fast and the overhead small, our solution method should work well in practice.

6.3.1 Convergence of SA

The cooling schedule, the repetition schedule and the candidate solution generator from Algorithm 3 in section 3.4.1 are the variables that together with the solution space determine the convergence rate.

Two different candidate solution generators were tested. The cooling schedule and the repetition schedule where used to control the number of SA invocations.

The bandwidth benchmark were run with messages of $2^{16} = 65536$ bytes, resulting in the solution space of Figure 4.2 and the improvement potential of the "ping-ping" benchmark in Figure 6.1.

The results from this convergence test is seen in Figure 6.7.

Baseline is the performance of the default routine selection within Scali MPI Connect, while the "Full randomization" and "Heap and temperature dependent" lines illustrate the performance of our solution method with different candidate generators used in the simulated annealing. The solution space was small enough to do an exhaustive search for the global optimum.

Starting at an initial solution at 0 SA invocations, the heap and temperature dependent candidate generator converge fast toward the optimal solution, while the fully randomized generator use a little more time. The difference between these two generators are more clearly seen when the solution space increase in size.

To illustrate this, a second benchmark program was run. It is similar to the first, but extended to contain four MPI_Send - MPI_Recv pairs instead of one. The solution space is now $35^8 \approx 2.25 * 10^{12}$ elements. This is too large to find the global optimum with an exhaustive search within reasonable time. Figure 6.8 show that the convergence speed of the completely random generator is relatively much slower than the heap and temperature dependent generator when the solution space increase.

6.3.2 Overhead of SA

Given the benchmarks above, we could also measure the overhead of the simulated annealing procedure. The first benchmark varied only one variable to generate a candidate solution and maintained

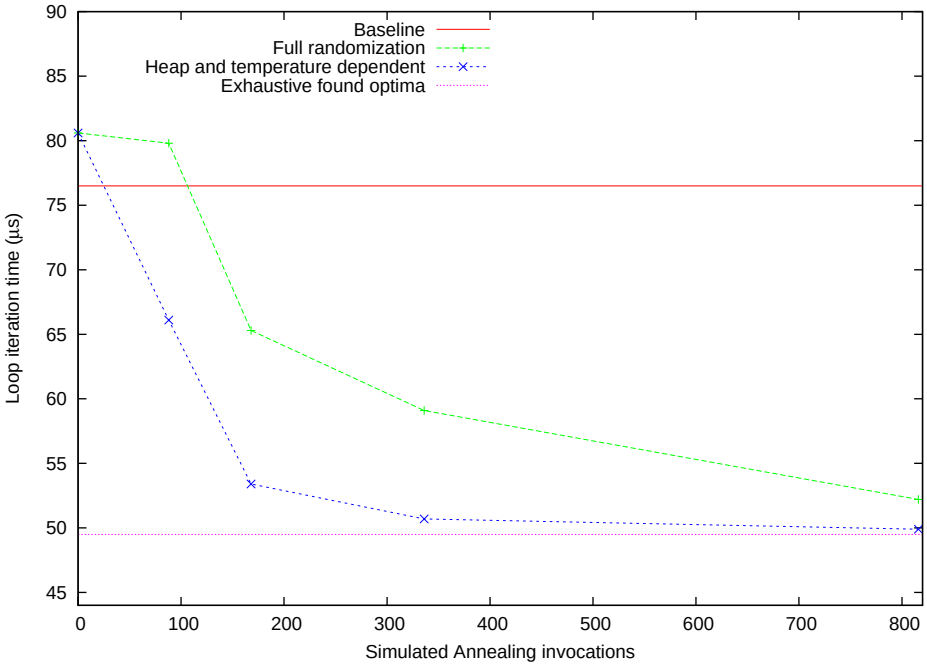


Figure 6.7: Convergence speed of SA on the solution space from Figure 4.2

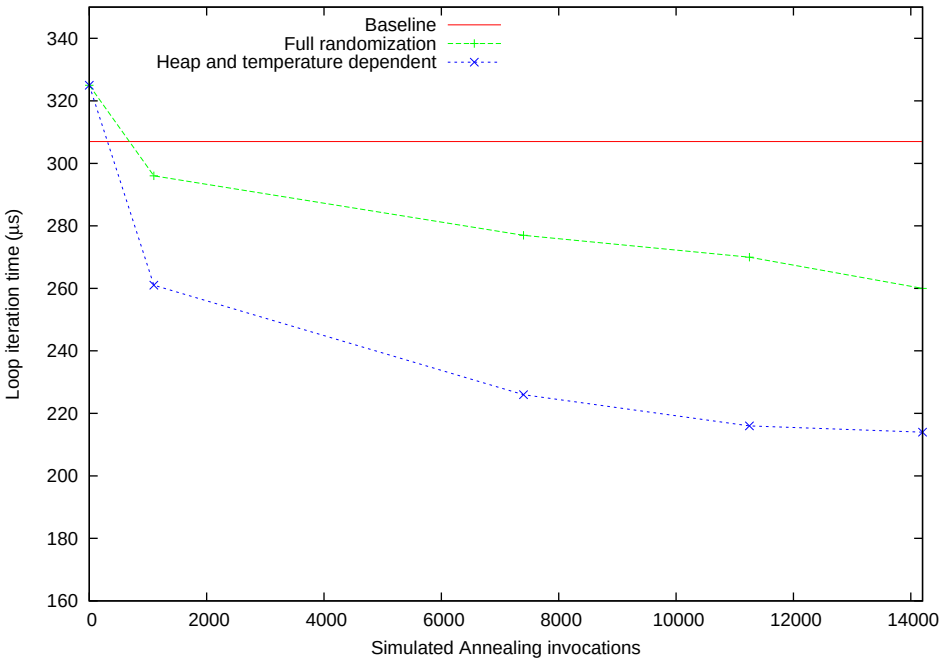


Figure 6.8: Convergence speed of SA on a 2250 billion element solution space.

only one heap per SA invocation. All solutions where set equal by using 35 equal memory copy routines, so the loop performance should remain constant.

Utilizing Equation (3.7), we find the overhead to be in the range

$$1.006\mu s < \text{Average SA overhead} < 1.766\mu s$$

with 5% confidence. This interval is suprisingly hard to determine in practice. The benchmark did 80000 loop iterations and it was tested with and without 80000 invocations of SA. A total of 240 measurements was necessary to calculate this interval. With 60 measurements, the interval contained 0, thus there was no statistical significant indication that the SA routine introduced any overhead.

In the second benchmark, four variables are changed in order to generate a candidate. Also, four heaps are maintained at each SA invocation. The overhead should scale linearly. However, the variance of the measurements were too large to determine a proper confidence interval. The difference between the sample means, however, gave an average SA overhead of $2.17\mu s$.

The lesson to be learned from this is that proper performance evaluation of parallel programs require a large amount of measurements, and the overhead introduced by the SA code are negligible when the loop is sufficiently large. In this case, sufficiently large is a loop with one communication call transmitting 65536 bytes through shared memory.

6.4 NAS Parallel Benchmarks

NAS-CG and NAS-MG benchmarks use point-to-point communication, and are supposed to mimic real-world applications in communication and computation. Consequently, an improvement of the execution time of these benchmarks would should verify whether our solution method would increase the performance in practical applications.

Both NAS benchmarks were of problem class B and were run on four and eight processors on the 4 AMD Opteron processor SMP and 16 dual-core Pentium III processor cluster respectively.

Problem class B introduce only a few iterations of the loops in the NAS benchmarks seen in Figures 5.2 and 5.1. As a consequence of this, the state of the simulated annealing was recorded after every execution and re-used in the next execution. Multiple executions were necessary so that the simulated annealing could be able to converge. The following measurements are done after the simulated annealing had "cooled" to the best solution found.

6.4.1 NAS-MG

NAS-MG has three reducible loops to optimize. Given four processes, these loop consists of one and two MPI_Send calls and one MPI_Irecv call. The data transmitted range from 8 to 131072 bytes, hence a potential performance improvement is possible as the communication pattern of this loop resemble the one seen in the "ping-ping" Figure 6.1.

Table 6.1 summarize the measurements. The dynamic optimization introduce a slight performance penalty compared to the standard. Also, the confidence interval is larger due to the random nature of the optimization heuristic.

Confidence	Standard [s]	Dynamically optimized [s]
95%	(19.738, 19.741)	(19.766, 19.774)
99%	(19.738, 19.742)	(19.764, 19.775)
99.9%	(19.737, 19.742)	(19.762, 19.777)

Table 6.1: Confidence intervals for NAS-MG performance.

An equal conclusion can be drawn from the measurement results when eight processors were utilized. The variance introduced with eight processes and ethernet interconnect hide much of difference between the two alternatives, however, using the statistic method ANOVA, we can conclude that the standard is the better alternative with 90% confidence.

6.4.2 NAS-CG

The NAS-CG benchmark has properties that suit our solution method better than the NAS-MG benchmark. There are three large, reducible loops, and the message sizes are 56000 bytes in problem class B. According to all the benchmark figures above, there exists a potential performance increase.

The cooling process done in advance for these measurements required between 10 and 100 executions, depending on the cooling scheme selected.

Confidence	Standard [s]	Dynamically optimized [s]
95%	(93.398, 93.435)	(93.571, 93.631)
99%	(93.387, 93.446)	(93.560, 93.641)
99.9%	(93.378, 93.455)	(93.547, 93.655)

Table 6.2: Confidence intervals for NAS-CG performance.

The confidence intervals obtained by the measurements of the NAS-CG benchmark are summarized in table 6.2. As with the NAS-MG benchmark there is a slight increase in execution time when the dynamic optimization is used and the confidence intervals are larger.

By utilizing eight processors there is less clustering of the measurement results leading to larger confidence intervals, but we can draw the same conclusions as with the NAS-MG benchmark. By using ANOVA, we find that with 90% confidence the standard method is the better alternative.

The slight performance reduction observed with the NAS benchmarks may come from different sources:

1. The overhead introduced by our solution method. There is an overhead in the registration of the return address, the look-up of loops and stop-addresses, and most significant, the overhead of the simulated annealing.

2. The loops in the NAS benchmarks call MPI_Irecv prior to calls to MPI_Send and finish with a call to MPI_Wait. Using shared memory communication, the intra-copy calls were mixed for the send and receive operations. The immediate consequence is that the sequence of intra-copy calls our solution method try to optimize is rearranged and not correctly optimized.

6.5 Discussion

In this Master's Thesis, our solution method is shown to improve the performance of parallel benchmarks through run-time optimization of the intra-copy routine selection.

Through the benchmark results presented in this chapter, we find that run-time optimization is necessary because of data reference and layout dependencies.

We also find that the simulated annealing part of our solution method converge relatively fast, and given the bandwidth benchmark, it will find an close-to-optimal solution. The results from the NAS benchmarks show that our solution method will not introduce a large penalty when it fails to improve performance.

The performance improvement of our solution method depend on the improvement potential given by the difference between optimal routine selection and the standard routines used in Scali MPI Connect, as well as the frequency and sequence of the MPI calls which in turn call the intra-copy routines. Further testing with other benchmarks and different systems would emphasize their characteristics rather than properties of our solution method.

6.5.1 Feasibility

The fact that our method work so well on the bandwidth benchmarks is captivating, but it should be useful in practice.

Portability

The code developed is rather flexible in the sense that it only need unique signatures as input to detect and recognize loops, and means to modify variables that influence the execution time of the loops in order to work as intended. There are no direct architectural dependencies, and it is in fact not a requirement that the code is utilized within an MPI implementation.

This make the code portable, although some implementation is most likely required to make it useful when put in a new context. The use of gcc's $\_\_builtin\_return\_address()$ require use of the gcc compiler, and a suitable timer must be available.

Usability

Our method is not fully transparent to the user. When applied to an application like the NAS kernels, several executions are necessary before the optimization heuristic converge. Information about the loops and the solution space has to be stored and re-used for each execution, typically controlled by a user. Also, generating a flowgraph from a "dry run" has to be managed by a user in the current implementation. This is not necessary, but practical for testing purposes.

Many applications are run with equal or close to equal data sets numerous times. For example, numerical weather prediction applications run with close to equal input sets multiple times every day. Applications like that are well suited for this solution method.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The goal stated in the introduction was to improve the execution speed of MPI applications based on their run-time characteristics. We have presented, implemented and tested a method for doing this.

We have shown that loops of library calls can be correctly recognized both at run-time and through loop detection algorithms used on a constructed flowgraph. Knowledge of these loops can then be exploited to optimize variable values during execution.

A version of simulated annealing has been used as a global optimization heuristic. It prove to work well in practice even with a dynamically evolving solution space, and a strict requirement to its execution speed. Although, it is not trivial to select proper parameters to obtain fast convergence.

The method in overall introduce very little overhead. The improvement of an application depend on a multitude of factors, but with the relatively small overhead of the method it can safely be applied to most applications without risk of performance degradation, only with a chance of improvement.

7.2 Future Work

The implementation and testing of the solution method touch many aspects that may be investigated further. The main issues and loose ends that can be picked up for further work are described here.

Both dynamic and static loop detection suffer from some weakness. The dynamic approach require more data structures and a higher instruction overhead than the static approach. It also require confidence in the loops found, but is able to detect the irreducible loops that static loop detection fail to find. These two approaches could be combined by using the flowgraph constructed by the static approach to find reducible loops and speculate the irreducible loops in the same way as the dynamic approach.

For optimization, a combined version of simulated quenching and probabilistic tabu search has been used. This heuristic rely on numerous variables that influence its performance. For example, there are

eight variables in the implementation that in combination regulate the probability function of Equation 3.3, and there are ten neighbor solution generators implemented yielding various performance properties. Proper tuning of simulated annealing is a field of research itself. A favorite for increasing convergence speed while retaining the ability to find a global optimum is adaptive selection of neighborhood generators based on characteristics of the convergence rate and solution space.

Other optimization techniques such as tabu search or metropolis may as well be tested.

Our testing has been limited to homogeneous clusters, applications with deterministic control flow, and no background load. All of these effects would be interesting to observe, but probably difficult to quantify correctly. Dynamic selection of internal algorithms should adapt to these environments.

There are many other internal MPI variables that may be tuned dynamically. Collective communication algorithms are a good candidate. However, it would require strict control of which processes that are involved in each call.

As mentioned in Section 6.5.1, it is not a requirement that this optimization is done in a parallel environment. In fact, all programs that contain loops where some work is done whose performance are influenced by some variables may benefit from our solution method.

Appendix A

ANOVA

The confidence-interval approach is considered by statisticans to be a relatively weak method of comparison, especially with more than two alternatives. Analysis of variance (ANOVA), however, is considered more robust and lead to more credible conclusions.

ANOVA divides the total variation between measurements into two separate components: one for random fluctuations, and one for actual differences among the alternatives.

The variation observed within one alternative is assumed to be due to measurement errors, while the variation between alternatives may be both measurement errors and difference between the alternatives. ANOVA quantify both values, and bring means to determine whether one alternative is statistically better than another.

ANOVA is pretty straightforward and can be summarized in a few points:

1. Given k different alternatives, make n measurements on each of them.
2. Organize all kn measurements in a table. One column for each of the alternatives.
3. Calculate column means:
$$\overline{y}_{.j} = \frac{\sum_{i=1}^{n} y_{ij}}{n}$$
These are the means for each of the alternatives.
4. Calculate overall mean:
$$\overline{y}_{..} = \frac{\sum_{j=1}^{n} \sum_{i=1}^{n} y_{ij}}{kn}$$
5. Write each measurement as a sum of the column mean and a deviation: $y_{ij} = \overline{y}_{.j} + e_{ij}$
6. Write each column mean as a sum of the overall mean and a deviation: $\overline{y}_{.j} = \overline{y}_{..} + \alpha_j$
7. Substitute the above notation of the column mean into all measurements:
$$y_{ij} = \overline{y}_{..} + \alpha_j + e_{ij}$$
8. Split the measurements into two separate components:
   (a) The variation due to the effects of the alternatives (SSA)

(b) The variation due to the errors (SSE)

9. Calculate these variations and their *variances* s_a^2 and s_e^2. Finally do an F-test.

$$
\begin{aligned}
SSA &= n\sum_{j=1}^{k}(\overline{y}_{.j} - \overline{y}_{..})^2 \\
SSE &= \sum_{j=1}^{k}\sum_{i=1}^{n}(y_{ij} - \overline{y}_{.j})^2 \\
s_a^2 &= \frac{SSA}{k-1} \\
s_e^2 &= \frac{SSE}{k(n-1)} \\
f &= \frac{s_a^2}{s_e^2} \\
f_{critical} &= F_{[1-\alpha;(k-1),k(n-1)]}
\end{aligned}
$$

10. If the computed f value is larger than the $f_{critical}$ value we can conclude that, at the α level of confidence the differences among the systems are statistically significant.

Appendix B

Code

The code written is part of this work and is attached as source files. The code is, however, somewhat premature and is not written as "production code". There is an overall lack of comment and structure in many parts of the code, as it was constructed to determine the properties of our solution method and not to be used directly by others.

Specifically, the attached code include the following files:

ob_inter.c The construction of flowgraphs, and usage of the intervals derived from flowgraphs is performed by the functions in this file.

ob_inter.h Give an interface to the global functions defined in ob_inter.c as well as defining code to detect return addresses.

ob_sao.c The functions in this file perform the simulated annealing.

The above files was compiled as a part of the Scali MPI Connect code. Calls to function prototypes in ob_inter.h were placed in all $PMPI$ function calls, as well as in the FORTRAN wrapper functions.

Given a flowgraph, its loops and intervals were found by a java code implementing the algorithms presented in Chapters 3 and 4. These java files are included in the attachment in the directory named "Loop detection code". The main method is found in the Identify.java file, which take a flowgraph represented by an adjacency matrix as input and produce output which in turn can be read by functions in ob_inter.c.

Bibliography

[AMD04] AMD. *Software optimization guide for AMD Athlon 64 and AMD Opteron processors*. Advanced Micro Devices, 2004.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[CHK01] K. Cooper, T. Harvey, and K. Kennedy. A simple, fast dominance algorithm, 2001.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[dAK01] M. de Alba and D. Kaeli. Runtime predictability of loops, 2001.

[ea53] Metropolis et al. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[For94] Message P Forum. Mpi: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[Fox93] Bennett L. Fox. Integrating and accelerating tabu search, simulated annealing, and genetic algorithms. *Ann. Oper. Res.*, 41(1-4):47–67, 1993.

[FY05] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of mpi collective communication routines. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402, New York, NY, USA, 2005. ACM Press.

[GKR94] V. Granville, M. Krivanek, and J.P. Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):652–656, 1994.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[HJJ03] Darrall Henderson, Sheldon Jacobson, and Alan Johnson. The theory and practice of simulated annealing. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 287–321. Kluwer Academic Publishers, 2003.

[HU74] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974.

[HZKK06] Chao Huang, Gengbin Zheng, Laxmikant Kalé, and Sameer Kumar. Performance evaluation of adaptive mpi. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 12–21, New York, NY, USA, 2006. ACM Press.

[Ing89] L. Ingber. Very fast simulated re-annealing. *Mathl. Comput. Modelling*, 12(8):967–973, 1989.

[Ing93] L. Ingber. Simulated annealing: Practice versus theory. *Mathl. Comput. Modelling*, 18(11):29–57, 1993.

[Int04] Intel. *IA-32 Intel architecture optimization: reference manual*. Intel Corporation, 2004.

[KYL03] Amit Karwande, Xin Yuan, and David K. Lowenthal. Cc–mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 95–106, New York, NY, USA, 2003. ACM Press.

[Lil00] David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, New York, NY, USA, 2000.

[Mos93] Pablo Moscato. An introduction to population approaches for optimization and hierarchical objective functions: a discussion on the role of tabu search. *Ann. Oper. Res.*, 41(1-4):85–121, 1993.

[Nat05] Thorvald Natvig. Auto-optimizing mpi programs. Master's thesis, NTNU, 2005.

[NPSC] Rajesh Nishtala, Neil Patel, Kaushal Sanghavi, and Kushal Chakrabarti. Automatic tuning of collective communication operations in mpi.

[OM96] Hirotaka Ogawa and Satoshi Matsuoka. OMPI: Optimizing MPI programs using partial evaluation. 1996.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992.

[Sca] Scali. Scali MPI Connect, http://www.scali.com.

[SGL96] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, 1996.

[SL03] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[StGDC03] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*, 2003.

[Tar73] Robert Tarjan. Testing flow graph reducibility. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107, New York, NY, USA, 1973. ACM Press.

[TG98] Jordi Tubella and Antonio Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *HPCA*, pages 14–23, 1998.

[TKN+] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D. Panda. Exploiting nonblocking remote memory access communication in scientific benchmarks on clusters.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[WMM02] R. E. Walpole, R. H. Myers, and S. L. Myers. *Probability and Statistics for Engineers and Scientists*. Prentice-Hall, Inc., New Jersey, seventh edition, 2002.