# A parallel solution for systems of integral equations

Marcelo Pasin[1]*, Edson Luiz Padoin[2]†
[1] Laboratório de Sistemas de Computação
Universidade Federal de Santa Maria (UFSM)
UFSM Campus, 97105-900, Santa Maria, RS, Brazil
pasin@inf.ufsm.br
[2] Universidade Regional do Noroeste do Estado do RS (UNIJUÍ)
Rua São Francisco, 501, 98700-000, Ijuí, RS, Brazil
padoin@inf.ufsm.br

*Abstract*—

**This paper presents a parallelization of a numeric method for solving systems of integral equations. The algorithm was originally developed to find transitions of superconductor properties based on environment conditions. The sequential numerical method presents a wide range of parallelization levels, with irregular processing costs. A parallel implementation of it has the oportunity of adapting the parallel grain, compromising load-balancing and communication, and achieving better efficiency.**

*Keywords*— **Parallel programming, irregular problems, load balancing, numerical methods, integral equations.**

## I. Introduction

This document describes the work on parallelizing a numeric method for solving systems of integral equations. More specifically, a sequential program implementing a numerical method was parallelized. The parallel version had to be load-balanced due to the irregularity of the original numerical method.

The sequential program used as basis of parallelization is intended to locate phase transitions of properties of superconductors. The problem resumes into a four-equations-four-variables system, all being multi-integral equations. The equation system is solved numerically using the fixed point method [Jer99]. The integrals are calculated by the Simpson method. In a regular execution, nearly one hundred billion integrations ($10^{11}$) are calculated.

Few are the data dependencies in the sequential algorithm, giving plenty of parallelizing oportunities. During the parallelizatrion process, they were exploited in different manners, to achieve better performance. One problem faced was that most of the parallel tasks obtained from the parallelization of the algorithm were irregular, which lead to load-balancig problems. This paper will present the experience done in parallelizing such algorithm, the load balancing decisions made, and the results obtained.

*Partly supported by FAPERGS grants 99/1492.5 and 00/2434.1.
†Partly supported by a FATEC grant.

## II. The sequential algorithm

The goal of the application used in the parallelization presented in this paper is to find points to plot a graph of phase transitions of superconductors [MS00]. In this graph, three environmental properties are in the axes: $H$, the applied magnetic field, $T$, the temperature and $G$, the strength of the pairing interaction. Three surfaces in the $(H, T, G)$ space are calculated, representing the transitions between three phases of interest [MS00]:

- the phase where there is a long range order corresponding to formation of pairs,
- the normal paramagnetic phase, and
- the spin glass phase.

Four variables have to be calculated in order to determine the phase of the matter: $\eta$ (superconductivity long-range order), $Q_z$ (spin glass long range order parallel to $H$), $R_z$ (susceptibility parallel to the magnetic field) and $R$ (susceptibility transversal to the magnetic field). This variables values are expressed in equations that are dependent of each others' values, forming a four-variable system of equations. The equations modeling the system are listed below.

$$\eta = \int_{-\infty}^{+\infty} D(w)\frac{\sinh(\beta\mu')}{I_\beta}$$

$$Q_z = K_q \int_{-\infty}^{+\infty} D(w)\left[\frac{\int_0^{+\infty} uD(u)\int_{-\infty}^{+\infty} D(v)S_\beta\theta}{I_\beta}\right]^2$$

$$R_z = \frac{1}{K_r}\int_{-\infty}^{+\infty} D(w)\frac{\int_0^{+\infty} uD(u)\int_{-\infty}^{+\infty} vD(v)S_\beta\theta}{I_\beta}$$

$$R = \frac{1}{4}\int_{-\infty}^{+\infty} D(w)\frac{\int_0^{+\infty} u^3D(u)\int_{-\infty}^{+\infty} D(v)S_\beta}{I_\beta}$$

where

$$I_\beta = \int_0^{+\infty} uD(u)\int_{-\infty}^{+\infty} D(v)\left[\cosh(\beta\mu') + \cosh(\beta|\vec{h}|)\right]$$

$$D(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}\,dx$$
$$K_r = 2\sqrt{(R_z - Q_z)}$$
$$K_q = \beta^2 J^2/2$$
$$S_\beta = \sinh(\beta|\vec{h}|)/(2\beta|\vec{h}|)$$
$$\theta = h_z/J$$
$$\beta = 1/T$$
$$\mu' = \sqrt{\beta\mu^2 + \Delta^2}/\beta$$
$$\Delta = \beta G\eta/2$$
$$h_z = J\left[v\sqrt{2(R_z - Q_z)} + w\sqrt{2Q_z} + \frac{H_z}{2J}\right]$$
$$h_+ = J\sqrt{2R}u_+$$
$$h_- = J\sqrt{2R}u_-$$

$J$ and $\mu$ are constants.

The algorithm used to calculate the points of the phase transition graph is presented in figure 1. A few hundreds of points must be found in order to plot a graph. The *main* procedure is currently being done by the user, because it is difficult to choose good intervals and steps for the *find-transition* procedure. Work is under way to develop a good algorithm to choose intervals. In any case, every one of these phase transition points can be found in parallel.

```
main procedure:
  i = 0
  while not enough points
    find-transition η[i], Q_z[i], R_z[i], R[i]
    i = i + 1
  plot surfaces
```

Figure 1. The *main* procedure (sequential)

In order to find a phase transition point (figure 2), the four-equation system is solved around ten thousand times, for different values of an interval of $H$, $T$ and $G$. Every solution has a different phase associated with it. Comparing them it is possible to find the values for which there is a phase transition point. All the steps of an interval, i. e. all the calls to the *solve* procedure, can be calculated in parallel.

In the *solve* procedure (figure 3), an iteration is done to calculate the values of the four integral equations until they converge to a solution (fixed point method). Although the sequential evaluation of the four equations result in faster system convergence, all the four equations can be calculated in parallel.

Last but not least, the equations are all integrals of integrals of integrals. In any level chosen, a parallel integration method can be used.

So, the parallelization levels found in the algorithm are:

- finding $N$ transition points;

```
find-transition procedure:
  k = 0
  for H = H_i to H_f step s_H
    for T = T_i to T_f step s_T
      for G = G_i to G_f step s_G
        solve η[k], Q_z[k], R_z[k], R[k]
        k = k + 1
  find j such as there is a phase transition
       in η[j], Q_z[j], R_z[j], R[j]
  return η[j], Q_z[j], R_z[j], R[j]
```

Figure 2. The *find-transition* procedure (sequential)

```
solve procedure:
  repeat
    Eta = ∫ f_η(η, Q_z, R_z, R, H, T, G)
    Qz  = ∫ f_Qz(η, Q_z, R_z, R, H, T, G)
    Rz  = ∫ f_Rz(η, Q_z, R_z, R, H, T, G)
    R   = ∫ f_R(η, Q_z, R_z, R, H, T, G)
  until η, Q_z, R_z and R converge
```

Figure 3. The *solve* procedure (sequential)

- solvind $N$ equation systems to find a transition;
- calculating the value of $N$ (4) equations; and
- calculating $N$ values of a function to integrate it.

No matter which level the parallelization should be chosen, any of the $N$ parallel tasks are irregular, because:

- there is no warranty to find a transition in every interval, so many intervals may have to be chosen until a transition is found;
- the number of iterations to solve a system depend on the quality of the initial values;
- the four integral functions are different; and
- the number of sub-intervals used in the integration of a function depend on its pitch.

Although the work described here focused only at parallelizing the sequential program, progress can still be done in the sequential method itself. The numerical algorithm is very expensive and research to improve it is under way. Nevertheless, all the sequential implementations made so far used both fixed point and Simpson methods. All these implementations follow the same scheme and can be used as basis of the parallelization proposed here.

## III. The parallelization

Irregularity appears at every level of parallelism of the application, as presented in the previous section. Load balancing must be taken into account because all computations have different execution times. Adequate frameworks must be chosen for (a) the parallel decomposition of the work to be done and for (b) the execution of the decomposed work.

One simple framework that suits well in this case is the processor pool model [WA00]. In this model, the algorithm of the program to be parallelized is recursively broken into smaller pieces that can be executed in parallel, called **parallel tasks**.

Every task must be registered in order to ask for its execution. In practice, an executing task divide its work into many smaller tasks and register them, respecting the data dependencies.

Many sequential producer-consumer programs, called **nodes**, are executed by the processors of the parallel machine. Every processor executes only one node. A node tries to acquire a registered task to execute as soon as it becomes idle. When a node acquires a task it starts to execute it, and becomes idle again after the execution.

Initially, one task is registered (the main procedure) and all processors are idle. The work is carried out in pieces by all the processors in a concurrent way.

Care should be taken in the task acquisition when they are irregular, is such a way that all processors receive the same amount of work. This can be better done if the all tasks are previously registered and have a known workload. In the algorithm presented, all tasks are previously unknown, different of each other, and have unpredictable workloads. In this case, an adaptive load balancing method must be chosen.

It is easy to recognize that if the load balancing algorithm is centralized, the processor pool model can be simplified to a master-slave model, being the master responsible for the load balancing. For a first implementation of this parallel algorithm it was decided that the load balancing algorithm should be simple and central. It should be kept to a minimum, in order to minimize the parallel overhead introduced by it.

### A. Parallel programming environment

A parallel machine to execute the algorithm had to be chosen. In order to minimize the need for special hardware, the authors decided to use a network of workstations as a virtual parallel machine. This could be done because of the low amount data used by the parallel tasks, requiring less network traffic. A classroom with 20 personal computers was allocated for this project's private use during the hours this application was executed. All the computers had AMD K6-II processors at 450 MHz and 64 MB of RAM, running Redhat Linux version 6.2. They were connected by a private Fast Ethernet (100baseT) network.

The software used to implement the parallel algorithms had to be well adapted to such a virtual parallel machine, and MPI [For93] or PVM [Sun90] would be the standard choices. Using either MPI or PVM is was impossible because most of their implementations are not suited for multithreaded use (not thread-aware). The Athapascan-0 communication em multithreading runtime library [PGBP97] was chosen because:

- it was conceived to run on networks of workstations and,
- it supports multiple cooperating threads on every parallel process,
- it supports remote procedure calls.

All 20 available computers were used as processing nodes for the application. At a time, each node ran only one heavyweight process (Unix process) and many lightweight processes (Unix threads).

### B. Simple master-slave model

Two initial implementations of the *find-transition* procedure (figure 2) were done to measure the overhead imposed by the master-slave model. The first implementation, called predistribute, previously distributed identical portions of the work to all the processors. This implementation had no master-slave overhead, as all work is previously distributed. In the second implementation, called naïve,[1] a master processor distributed the work to the slave processors as they become available.

To avoid load-unbalancing problems, a regular interval of $H$, $T$ and $G$ was chosen, with 40 steps and no transition in it. In both implementations only the *find-transition* procedure was executed. The calls to *solve* for every step were done in parallel and all the parallel calls returned in roughly the same time. The results are shown in figure 4. The overhead of naïve over predistribute was nearly constant and close to one second per step.

### C. The benchmark interval

A certain irregular interval was chosen as a benchmark, with a large degree of irregularity. $H$ and $T$ were respectively fixed in 0.25 and 1.0 and $G$ ranged from 4 to 11.9 with 0.1 steps (80 different values). One should note that all the intervals that contain phase transitions (i.e. all the intervals that matter) present the same order of irregularity. The figures 5 and 6 present respectively the sequential time to execute and the number of iterations needed to converge on every step. The total amount of time spent in a sequential execution of all the steps in any of the available computers was 33370 seconds (a little more than 9 hours).

The naïve implementation was then used to execute the

[1] For naïve load balancing.

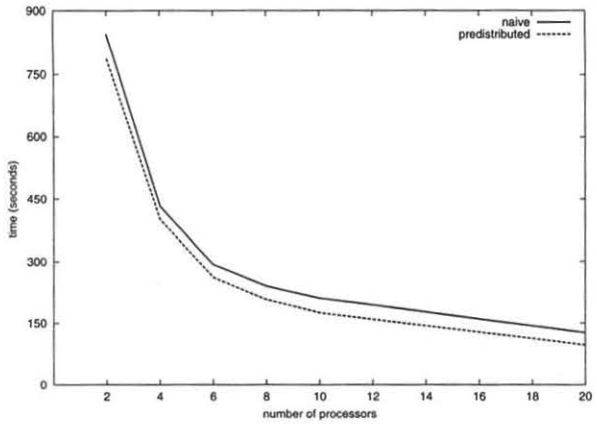


Figure 4. Master-slave overhead

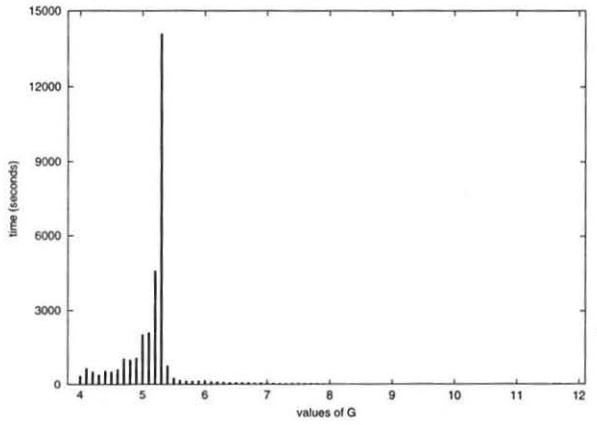


Figure 5. An irregular interval, execution time of the steps

benchmark interval on 10 nodes. The algorithm delivered the first 10 steps to the nodes and, as the nodes finished their work, another steps were allocated to them until all ended. One first execution (not shown here) yielded an execution time of 23600 seconds. Of course, due to the simplicity of the load balancing algorithm, many nodes were idle for most of the time, but this was not the most important problem. Adding all the busy CPU times of the nodes totaled to 67916 seconds, far more than the sequential time. After a careful look, it was found out that the Athapascan-0 communication daemon was polling the network too often, spending nearly 50% of the CPU time.

To reduce the Athapascan-0 communication daemon interference, its priority was changed to the minimum allowed. This decision does not empoverish communication performance because the only moment where it is needed in the naïve implementation is when a node needs more work. In that moment the working threads are blocked and the communication daemon would inevitably run.

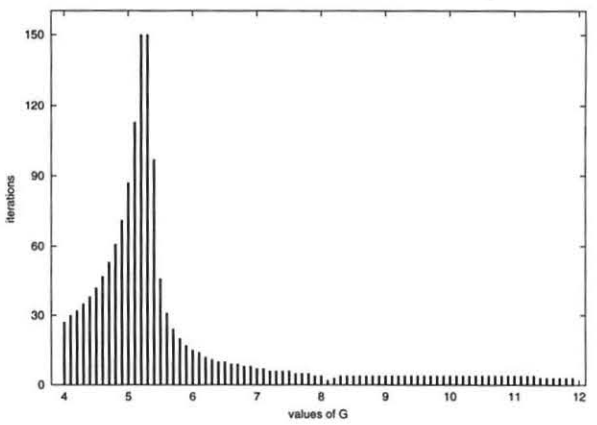


Figure 6. An irregular interval, iterations to converge of the steps

The same naïve implementation was run again on 10 nodes and better results were obtained. The execution times for every processing node are drawn in figure 7. The bars displayed represent the amount of processing done in every processor. Gray and black were used to differentiate the steps of the same node. Adding all the busy times of the processors gives 36655 seconds, but the last processor (node 3) became idle at 15589 seconds. Speedup is 2.14 (efficiency is 21.4%), still very low.

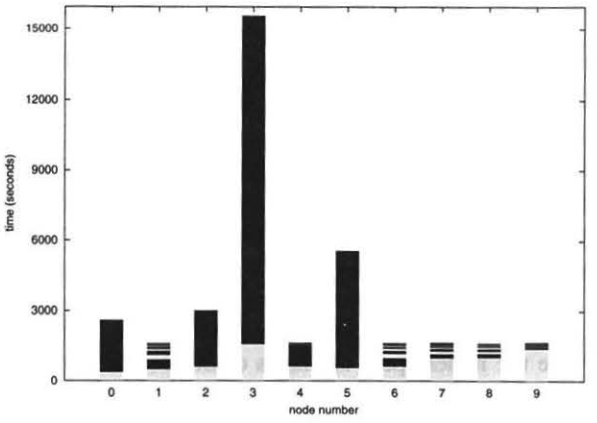


Figure 7. Master-slave, irregular interval, naïve load balancing

Visibly the load is unbalanced. Most of the nodes receive very small steps to process, while node 3 receives the step 5.3, that lasts much longer than others. Even if better load distributions could be found, none of them would be close to the ideal. The steps are too irregular. Worse, there are not enough steps to allow the balance in the long-term. The problem is that the granularity [KS88] chosen is too coarse.

Chosing another granularity means deciding to parallelize in a different level. For example, the parallelization could be

done at the integration level. The problem is that if too many integrations were calculated, the parallel overhead would be too large. A compromise situation should be devised, using coarse granularity while there are steps to distribute and re-distributing integration work when steps are over.

### D. The adaptive load balancing

In order to vary the granularity of the parallelization, the algorithm was changed to do as follows. While there are steps to work on, the master keeps giving those steps to idle slaves. When a slave becomes idle and no more steps are available to process, the master forwards the idle slave name to a busy slave node. The busy slave node, upon reception of the first slave name, starts to run a master-slave parallel algorithm for calculating integrals, using the slaves forwarded to it. It becomes a integral-master node. When an integral-master node ends its work it returns to the main master the slave names it received. The main master keeps forwarding idle slave names until there are no slaves nor steps to process. This algorithm is called adaptive algorithm.

In the current implementation of the adaptive algorithm, idle slaves are forwarded to the node with the oldest running step. This is based on the assumption that if a node is taking a long time to process its step it will keep doing so. One could say that if a node has already taken a long time it is probably reaching the end of its step. Although the first hypothesis was chosen, more study should be done on this subject in order to define which case better applies here.

Starting to process the integrals in parallel means that the granularity is taken to a finer level. One should notice that the equations have three levels of integrations for $Q_z$, $R_z$ and $R$. To avoid an excessively fine grain, which would have a large parallel overhead, only the first level of integration is done in parallel. This level could be changed in cases where the irregularity of the integrals is too large.

The figure 8 shows the results achieved. Gray and black bars represent the execution of steps. White bars show the time the nodes spent as slaves of Simpson integrations. With the adaptive algorithm, all nodes end their execution in around 4300 seconds. The speedup achieved is 7.74 (77.4% of efficiency). Nodes allocated with small steps mostly finish their work by 1700 seconds and become slaves for calculating integrals of the other nodes. By the time of 3000 seconds, only one node still have its step to process, all other nodes help it.

The adaptive algorithm was executed in a different number of nodes, to evaluate the evolution of its parallel overload. The figure 9 shows this evolution, by the means of the execution time and the speedup. The efficiency for 4-10 nodes is near to 80%, which can be seen as very good figure for this type of architecture. The efficiency gets lower outside this range.

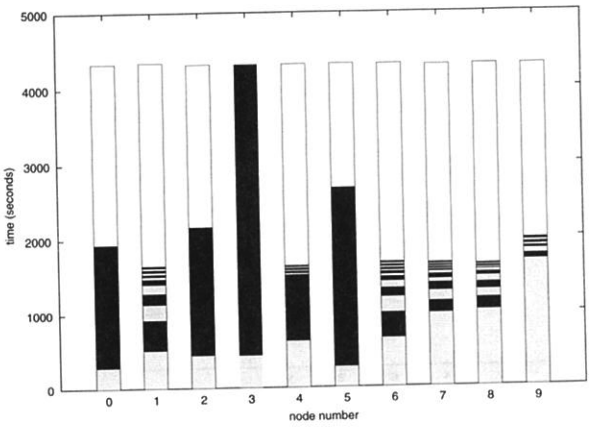


Figure 8. Master-slave, irregular interval, two-level load balancing

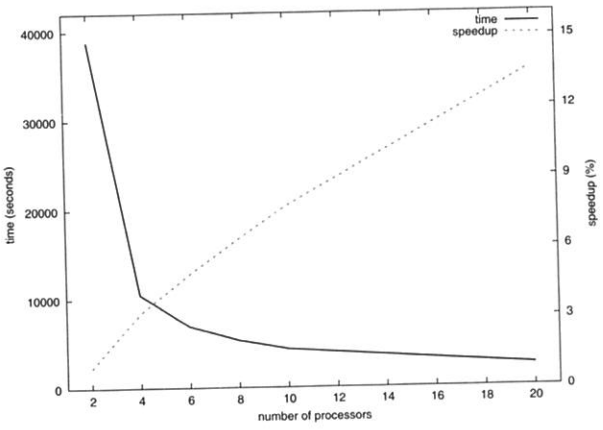


Figure 9. Parallel overload for adaptive algorithm

An evaluation was done in the algorithm in order to find out why efficiency gets lower when the number of nodes increases above 10. The Simpson integration was extracted from the *solve* procedure and executed alone in sequential and in parallel, for a number of nodes between 0 an 15. It can be seen in the figure 10 that the execution time for integrations are lower for 8, 9 and 10 processors. Breaking Simpson integrations in more than 10 pieces (to use more than 10 processors) requires more communication, yielding a slower execution in the network used. Work is also currently under way to better parallelize the Simpson integrations.

## IV. Conclusion and Future Work

The results presented here show that the paralellized algorithm has achieved good efficiency. The small amount of the communication makes it possible, even in an architecture with poor network performance. Better performances could be obtained with faster communication networks. Future implementations should improve the algorithms of the

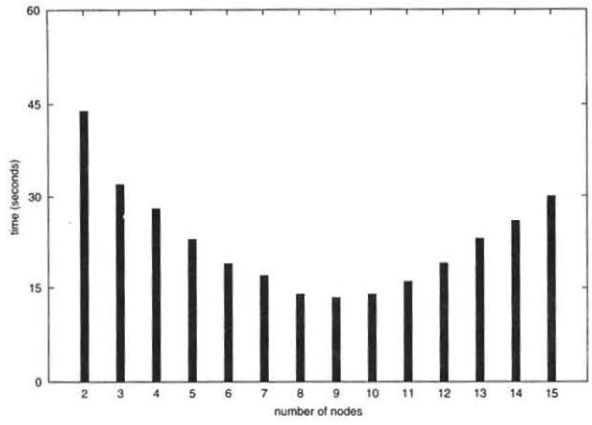


Figure 10. Parallel simpson execution time

fixed point and Simpson, that were not touched so far.

The centralized master-slave load balancing algorithm should be abandoned and a distributed implementation sould be adopted. This would reduce the current bottleneck of task allocation. One new implementation is under study, using migrating asynchronous remote procedure calls [ATK92, GRCD98].

Also, load balancing has again proven to be difficult and grain-dependent. The presented application has a wide range of choices for its granularity, which is not usually the case. Further study would be necessary to generalize the concept of grain adaptation [Cav99].

## ACKNOWLEDGEMENTS

This paper acknowledges the authors of the sequential algorithm, Alex André Schmidt, and of the physical problem, Sérgio Garcia Magalhães, respectively working at the the departments of Mathematics and Physics of UFSM. It would not be possible to produce this paper without their work and collaboration.

## REFERENCES

[ATK92] A. L. Ananda, B. H. Tay, and E. K. Koh. A Survey of Asynchronous Remote Procedure Calls. *sigops*, 26(2), April 1992.

[Cav99] Gerson-Geraldo-Homrich Cavalheiro. *Athapascan 1 : Interface genérique pour l'ordonnancement dans un environnement d'exécution parallèle.* Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, November 1999.

[For93] Message Passing Interface Forum. MPI: A Message Passing Interface. *Proceedings of the Supercomputing Conference*, pages 878–885, November 1993.

[GRCD98] François Gallilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mathias Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 88–95, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[Jer99] Abdul J. Jerri. *Introduction to Integral Equations With Applications.* John Wiley & Sons, 2nd edition, 1999.

[KS88] Clyde P. Kruskal and Carl H. Smith. On the notion of granularity. *The Journal of Supercomputing*, 1(4):395–408, August 1988.

[MS00] Sérgio Garcia Magalhães and Alex André Schmidt. Fermionic heisenberg model for spin glasses with BCS pairing interaction. *Physical Review B*, 62(17):686–693, November 2000.

[PGBP97] Marcelo Pasin, Ilan Ginzburg, Jacques Briat, and Brigitte Plateau. Athapascan runtime: efficiency for irregular problems. In *Proceedings of Euro-Par'97*, Aug 1997.

[Sun90] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, practice and experience*, 2(4):315–339, December 1990.

[WA00] Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel comuters.* Prentice-Hall, 2000.